



TECHNISCHE
UNIVERSITÄT
DARMSTADT

DISTRIBUTED COMPUTATION MEETS
BLOCKCHAIN:
ADVANCED CRYPTOGRAPHIC SERVICES FROM
BLOCKCHAIN FEATURES

Vom Fachbereich Informatik der TU Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doctor rerum naturalium (Dr. rer. nat.)

von

David Kretzler

Darmstadt 2024

Gutachter: Prof. Sebastian Faust, Ph.D.
Prof. Carmit Hazay, Ph.D.

Datum der Einreichung: 14.08.2024

Autor: David Kretzler

Titel: Distributed Computation Meets Blockchain:
Advanced Cryptographic Services from Blockchain Features

Ort: Darmstadt, Technische Universität Darmstadt

Datum der mündlichen Prüfung: 27.09.2024

Veröffentlichungsjahr der Dissertation auf TUprints: 2024

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
<http://tuprints.ulb.tu-darmstadt.de>
tuprints@ulb.tu-darmstadt.de

Bitte zitieren Sie dieses Dokument als:

URN: [urn:nbn:de:tuda-tuprints-286614](https://nbn-resolving.org/urn:nbn:de:tuda-tuprints-286614)

URI: <http://tuprints.ulb.tu-darmstadt.de/id/eprint/28661>

Urheberrechtlich geschützt / In Copyright (<https://rightsstatements.org/page/InC/1.0/>)

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

(D. Kretzler)

Wissenschaftlicher Werdegang

Oktober 2013 bis September 2016: Bachelor of Science in Wirtschaftsinformatik an der Dualen Hochschule Baden-Württemberg – Karlsruhe

Oktober 2016 bis Mai 2019: Master of Science in Informatik an der Technischen Universität Darmstadt

Oktober 2018 bis Oktober 2019: Master of Science in IT-Sicherheit an der Technischen Universität Darmstadt

Oktober 2019 bis September 2024: Doktorand am Lehrstuhl für Angewandte Kryptographie an der Technischen Universität Darmstadt bei Prof. Sebastian Faust.

Acknowledgments

Pursuing a PhD has been a challenging yet rewarding endeavor, full of diverse and exciting experiences. Having started my academic career with a bachelor's degree in business informatics from the practically oriented Cooperative State University, I could never have imagined that I would be doing a PhD in cryptography. Indeed, at the beginning of my studies, I had only a very vague idea of what cryptography was. Nevertheless, I am currently writing the final parts of my dissertation and would like to take a moment to look back and thank all the people who have supported me in this (probably last) step of my academic career.

First and foremost, I would like to thank my PhD supervisor, Sebastian Faust, for his continuous support and for giving me the freedom to choose my research direction according to my own interests. I am grateful for the constant academic and non-academic guidance, as well as the opportunity to attend various conferences, summer schools, and research visits. Second, I would like to express my sincere gratitude to Carmit Hazay, who, although not officially appointed, has been my co-supervisor throughout my PhD studies. Carmit has supported me not only from a professional point of view but also far beyond that. Besides others, she made it possible for me to spend six months in her research group and to find an internship at Intel Research. Here, I would also like to thank Ittai Abraham, who supervised me during my internship, and Thomas Schneider and István Zsolt for serving on my disputation committee alongside Sebastian and Carmit.

Naturally, I am also grateful to all my co-authors and colleagues for the fruitful discussions, the challenges we solved, the publications we produced, and the many great non-research moments we shared. I would like to mention Benjamin Schlosser and Rahul Satish by name. Working with Benni, with whom I did most of my projects together, was characterized by a rich exchange of ideas, mutual support, and excellent collaboration. Rahul, on the other hand, made me feel like part of the family during my research visit to Tel Aviv from the moment I first entered the lab. I am also very grateful to Dorothee Nikolaus and Jacqueline Wacker for their administrative support, which allows my colleagues and me to focus on research without worrying about the bureaucratic hurdles one must overcome while pursuing a PhD.

Last but not least, I owe my deepest gratitude to my parents, my brother, and

my friends, not only for their support and encouragement but also for the many times they provided me with the much-needed distraction from the daily grind of research.

List of Own Publications

Peer-reviewed Publications

- [86] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Financially Backed Covert Security”. In: *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part II*. 2022, pp. 99–129. **Part of this thesis.**
- [88] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Generic Compiler for Publicly Verifiable Covert Multi-Party Computation”. In: *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*. 2021, pp. 782–811. **Part of this thesis.**
- [90] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Putting the Online Phase on a Diet: Covert Security from Short MACs”. In: *Topics in Cryptology - CT-RSA 2023 - Cryptographers’ Track at the RSA Conference 2023, San Francisco, CA, USA, April 24-27, 2023, Proceedings*. 2023, pp. 360–386. **Part of this thesis.**
- [91] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Statement-Oblivious Threshold Witness Encryption”. In: *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*. 2023, pp. 17–32. **Part of this thesis.**
- [95] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A. Sadeghi. “POSE: Practical Off-chain Smart Contract Execution”. In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. 2023. **Part of this thesis.**
- [152] D. Richter, D. Kretzler, P. Weisenburger, G. Salvaneschi, S. Faust, and M. Mezini. “Prisma : A Tierless Language for Enforcing Contract-client Protocols in Decentralized Applications”. In: *ACM Trans. Program. Lang. Syst.* 3 (2023), 17:1–17:41. **Part of this thesis.**

Articles in Submission

- [93] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Threshold BBS+ From Pseudorandom Correlations”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1076.

My Contribution

The results presented in this thesis are based on six publications. The projects that led to these publications were collaborations between myself and several excellent researchers: Sebastian Faust, Tomasso Frassetto, Carmit Hazay, Patrick Jauerning, David Koisser, Mira Mezini, David Richter, Ahmad-Reza Sadeghi, Guido Salvaneschi, Benjamin Schlosser, and Pascal Weisenburger. In the following, I aim to specify my individual contribution to each of the publications included in this thesis.

Chapter 3 is based on [86, 88, 90], joint works with Sebastian Faust, Carmit Hazay, and Benjamin Schlosser. In [88], I came up with the initial idea, the utilization of time-lock puzzles to address shortcomings of prior work on publicly verifiable covert (PVC) security. Based on this idea, Benjamin and I jointly designed and specified our protocol. Subsequently, I conducted the security proof for our protocol, for which I received some support from Benjamin. Moreover, I provided extensions, allowing for shorter proofs of misbehavior and a more efficient time-lock puzzle generation. Finally, I compared the deterrence factor of our protocol to prior work. In [86], I designed and specified the transformations from different classes of PVC secure protocols to financially backed covert secure protocols, proved their security, proposed the single-gate validation, and conducted the implementation and evaluation. In [90], I came up with the initial idea to reduce the security of malicious secure online protocols to covert security by reducing the MAC length, while Benjamin observed that such a reduction benefits the TinyOT protocol. Benjamin and I both worked on the covert security composition theorem. I provided the rationale behind the theorem, identified the theorem’s constraints, and designed the simulator. Furthermore, I specified the covertly secure TinyOT online phase and proved its security. Finally, I designed, specified, and evaluated the cut-and-choose-based covert secure offline phase.

Chapter 4 is based on [91], a joint work with Sebastian Faust, Carmit Hazay, and Benjamin Schlosser. In this project, I contributed the initial idea of hiding the statements in witness encryption and proposed the applications. Benjamin defined the notion of statement oblivious threshold witness encryption (SO-TWE), while I defined the notion of oblivious threshold tag-based encryption (O-TTBE). Moreover, I designed, specified, and proved the security of the construction of SO-

TWE from O-TTBE, the instantiation of O-TTBE from bilinear pairings, and the construction of O-TTBE from anonymous threshold identity-based encryption.

Chapter 5 is based on [95], a joint work with Sebastian Faust, Tomasso Frassetto, Patrick Jauerning, David Koisser, Ahmad-Reza Sadeghi, and Benjamin Schlosser. All co-authors contributed to the discussions leading to the design of our smart contract platform. Benjamin Schlosser and I jointly designed and specified the protocol and proved its security. Additionally, I implemented the on-chain manager contract required by our platform.

Chapter 6 is based on [152], a joint work with Sebastian Faust, Mira Mezini, David Richter, Guido Salvaneschi, and Pascal Weisenburger. David Richter and I jointly designed the programming language Prisma. Moreover, I supported David Richter with the implementation of the language's compiler, implemented the case studies, and performed the evaluation.

Abstract

Today’s blockchain systems are no longer just about financial transactions within decentralized networks. Instead, they offer a wide range of additional features. A recent trend in cryptography leverages the rich functionality provided by blockchains to implement new cryptographic services and enhance existing ones. However, the potential of blockchain systems is far from exhausted and there is still significant room for improvement in existing blockchain-based cryptographic solutions. This thesis, therefore, aims to identify and unlock further potential for providing more advanced cryptographic services by identifying and closing gaps in prior work on blockchain-based cryptography.

Covert security, introduced by Aumann and Lindell (TCC’07), is a security notion for cryptographic protocols that allows an adversary to successfully cheat and break the protocol’s security with a fixed probability $1 - \epsilon$, while honest parties are guaranteed to detect the cheating attempt with probability ϵ . Zhu et al. (CCS’19) proposed strengthening this notion by financially punishing detected cheaters via a smart contract. However, their work focuses on a specific two-party protocol. This thesis advances their ideas by demonstrating how to transform an arbitrary semi-honest secure protocol into a financially-backed covert secure protocol combining cheating detection with immediate financial punishment.

Witness encryption, a primitive introduced by Garg et al. (STOC’13), allows a party to encrypt a message under a statement x from an NP-language \mathcal{L} with relation \mathcal{R} , such that the ciphertext can only be decrypted by a party knowing the corresponding witness w for which $\mathcal{R}(x, w)$ holds. Unfortunately, known instantiations of general-purpose witness encryption are based on strong assumptions and lack efficiency. Moreover, the standard notion of witness encryption does not consider the need to keep the statement used for encryption private. Goyal et al. (PKC’22) addressed the former shortcoming by demonstrating how a committee elected by a blockchain can provide a service equivalent to witness encryption but with significant higher efficiency and without the need of strong cryptographic assumptions. We advance on this idea by showing how such a committee-based approach to witness encryption can be adopted without disclosing the statement used for encryption, thereby addressing the latter shortcoming.

We envision a virtual trusted third party (V-TTP) as a service that is continu-

ously available, strictly adheres to expected behavior, keeps its state and communication secret, and is capable of performing complex computations. A promising approach to implementing a V-TTP is through a smart contract deployed on a blockchain. Smart contracts inherit excellent liveness guarantees from the underlying blockchain and ensure the correct execution of their code. However, traditional smart contracts are inherently public and limited in their complexity. While numerous proposals address these limitations, they often focus on only one aspect or introduce new shortcomings, such as requiring locked collateral. In this thesis, we propose a new smart contract platform that addresses the limitations of previous smart contract systems in one holistic solution.

Naturally, utilization and improvement of blockchain features go hand in hand. During our work with smart contracts, we identified several shortcomings in the prevalent approach to smart contract development. We address these shortcomings by proposing a new programming language for smart contracts, which reduces the risk of security critical programming errors and increases the usability of smart contracts.

Contents

1. Introduction	1
1.1. Financially Backed Covert Security	2
1.2. Statement Oblivious Witness Encryption	4
1.3. Virtual Trusted Third Party	6
1.4. Modern Programming Language for Decentralized Applications . .	7
1.5. Thesis Outline	8
2. Preliminaries	9
2.1. Notation and Convention	9
2.2. (Publicly Verifiable) Covert Secure Multiparty Computation	10
2.2.1. Covert Security	11
2.2.2. Publicly Verifiable Covert Security	13
2.3. Cryptographic Building Blocks	15
2.4. Blockchain and Smart Contracts	19
2.5. Trusted Execution Environments	22
3. Financially Backed Covert Security	24
3.1. Our Contribution	28
3.2. Key results	30
3.2.1. Generic Compiler for Publicly Verifiable Covert Security . .	30
3.2.2. Financially Backed Covert Security	34
3.2.3. Covert Security From Short MACs	41
3.3. Related Work	45
4. Statement Oblivious Witness Encryption	48
4.1. Our Contribution	49
4.2. Key Results	49
4.3. Related Work	53
5. Virtual Trusted Third Parties	54
5.1. Contribution	55
5.2. Key Results	55
5.3. Related Work	59

Contents

6. Modern Programming Language for Decentralized Applications	62
6.1. Our Contribution	63
6.2. Key Results	64
6.3. Related Work	66
7. Conclusion	69
8. Bibliography	72
Appendix A. Generic Compiler for Publicly Verifiable Covert Multi-Party Computation	92
Appendix B. Financially Backed Covert Security	123
Appendix C. Putting the Online Phase on a Diet: Covert Security from Short MACs	154
Appendix D. Statement-Oblivious Threshold Witness Encryption	182
Appendix E. POSE: Practical Off-chain Smart Contract Execution	199
Appendix F. Prisma : A Tierless Language for Enforcing Contract-client Protocols in Decentralized Applications	218

1. Introduction

The introduction of the first decentralized cryptocurrency, Bitcoin [145], in 2008 profoundly impacted research, industry, and society. It established a novel field of research, increased interest in decentralized systems across research and industry, and significantly raised the general public's attention to cryptography.

Blockchain-based cryptocurrencies comprise two fundamental components: a distributed database, known as the *ledger*, and a consensus protocol. The ledger stores all transactions within the system in a series of blocks. The order of transactions within a block and the order of blocks are fixed and immutable. The consensus protocol ensures that all involved parties share a consistent view of the ledger at all times and guarantees the immutability of blocks added to the ledger. This allows all parties to track every movement of assets and, hence, the balances of all parties. The primary advantage of a decentralized cryptocurrency compared to a traditional payment system is that the cryptocurrency does not rely on a trusted authority. Instead, it guarantees the correct execution of transactions as well as availability and censorship resistance as long as the majority of the maintainers are honest.

While decentralization is an appealing feature for digital payment systems, it does not come without costs. Most prominently, decentralized cryptocurrencies often suffer from limited scalability. For example, Bitcoin is restricted to just seven transactions per second [66], whereas the VISA network can process up to 65 000 transactions per second [168]. The challenges in designing decentralized cryptocurrencies, coupled with a growing demand for additional features, such as increased privacy, have spurred numerous advances in blockchain technology from both academic institutions and industry. Today, 15 years after Bitcoin initiated the blockchain era, there is a plethora of cryptocurrencies with a combined market capitalization exceeding 1.5 trillion USD [61]. Furthermore, blockchain systems are no longer only about payments in decentralized networks but offer a wide range of additional functionalities.

Most importantly, many modern blockchains are capable of executing Turing-complete programs, known as *smart contracts*, on the blockchain. Ethereum [83] introduced the concept of smart contracts, which have since been adopted by numerous other blockchain platforms, including [2, 15, 29, 52, 160, 166]. Smart

1. Introduction

contracts can own and transfer assets based on the programmed semantics. The consensus protocol ensures the correct execution of the deployed contract code and consistency of the contract state across the distributed database. The decentralized nature of the blockchain ensures liveness and provides censorship resistance. However, in many blockchain systems, such as Ethereum, the computation that can be performed within a single block is limited, making smart contracts inefficient and infeasible for extensive computations.

The rich set of functionality provided by modern blockchain systems sparked a new trend in cryptography (among others [3, 49, 57, 106, 107, 182]). Researchers are leveraging blockchain features to revisit established problems lacking efficient solutions and to tackle new problems previously deemed unsolvable. However, due to the relatively recent emergence of this trend, existing solutions still hold significant potential for improvement. Consequently, this thesis aims to identify and address gaps in prior work on blockchain-based cryptography, thereby providing more advanced blockchain-based cryptographic services. We will focus on the directions introduced in the following.

1.1. Financially Backed Covert Security

Secure multiparty computation (MPC) [104, 180] allows a set of parties to jointly compute an arbitrary function on private inputs without revealing anything but the output of the function. The privacy of the inputs and the correctness of the outputs are guaranteed even in the presence of an adversary who corrupts a subset of the parties. Traditionally, MPC protocols have been designed in either the semi-honest security model or the malicious security model. While the adversary in the semi-honest model is constrained to follow the protocol specification, the adversary in the malicious model can deviate arbitrarily from the protocol specification. Clearly, the malicious model provides stronger security guarantees. However, the higher level of security comes at a significant efficiency overhead.

As a middle ground between the two notions, Aumann and Lindell [14] proposed the notion of *covert security*. In covert security, the adversary is allowed to deviate from the protocol specification, as in the malicious model. The honest parties are guaranteed to detect cheating attempts with a fixed probability ϵ , referred to as the *deterrence factor*. If the detection fails, the adversary can break all security properties. The rationale behind this notion is that adversaries refrain from cheating due to the deterrent effect of reputational damage connected to the detection event. One shortcoming of covert security is that parties involved in the protocol cannot transfer knowledge about malicious behavior to external parties.

1. Introduction

In order to address this drawback, Asharov and Orlandi [12] proposed the notion of *publicly verifiable covert (PVC)* security. In PVC secure protocols, parties that detect a cheating attempt obtain a *proof of misbehavior (PoM)* that can be used to convince any external party, hereafter referred to as *judge*, about the malicious behavior of a corrupted party. This way, PVC security translates the reputational damage previously restricted to the parties involved in the protocol to the public.

However, there are instances where the public reputational damage of PVC security is still insufficient to deter cheating attempts, e.g., when parties hide behind replaceable pseudonyms without disclosing their true identity. Therefore, Zhu et al. [182] proposed to utilize a smart contract to link the reputational damage of a detected cheating attempt with immediate financial punishment. The naive approach is to run a PVC secure protocol between the involved parties but require them to lock a security deposit at the beginning of the protocol at a dedicated judging smart contract. In the event of detected misbehavior, the honest parties submit the received PoM to the smart contract, which verifies the PoM and withholds the deposit of the malicious parties. Unfortunately, this approach is not practical. The PoM verification within known PVC secure protocols [12, 70, 116, 126, 182] requires the judge to recompute large parts of the executed protocol. Such computation is not feasible for a resource-constrained smart contract. In [182], the authors present a protocol that follows the same blueprint as the naive approach, i.e., they run a PVC secure protocol and lock collateral at a smart contract, but augment the protocol with a highly efficient interactive punishment procedure. This allows the punishment to be realized via a smart contract. However, the authors only consider a specific two-party protocol and tailor their techniques to the specific underlying two-party PVC secure protocol.

Our contribution. In this thesis, we aim to broaden the scope of [182] by enhancing arbitrary semi-honest secure protocols with covert security and the financial punishment of detected cheating attempts. We call such protocols *financially backed covert (FBC)* secure protocols. As the financial punishment is enabled by smart contracts, we require highly efficient judging and punishment procedures to ensure that the judge can indeed be instantiated by a smart contract.

Note that FBC security builds directly on top of PVC security, as misbehavior needs to be publicly verifiable to be punishable by an external judge. Therefore, we envision a transformation from semi-honest security to FBC security to be composed of two steps, one from semi-honest security to PVC security and one from PVC security to FBC security. We innovate in both of these steps.

In [88], we focus on the first step and present a generic compiler transforming any semi-honest secure multiparty protocol into a PVC secure multiparty protocol.

1. Introduction

The only prior work [70] considering PVC security in the multiparty setting also presents a generic transformation from semi-honest security to PVC security but restricts the deterrence factor of the resulting PVC secure protocol by $\frac{1}{n}$ from above (where n denotes the number of involved parties). Our compiler addresses this limitation by proposing a new approach to PVC security based on time-lock puzzles that allows for arbitrary deterrence factors.¹

In [86], we consider the second step. On the one hand, we formally define the notion of FBC security. On the other hand, we present several compilers transforming different classes of PVC secure protocols into FBC secure ones. The FBC secure protocols generated by our compilers incorporate a highly efficient judging procedure, allowing the instantiation of the judge via a smart contract. We showcase the efficiency of the judging procedure in the resulting protocols based on a prototype implementation of the judge in the form of an Ethereum smart contract. The classes considered by our transformations comprise all previously proposed PVC secure protocols, including the ones generated by the compiler presented in our previous work [88].

Although not directly related to financial punishments, we have identified disregarded potential for efficiency improvement in certain classes of covert secure MPC protocols during our work on covert security. Therefore, we round off our contribution in the field of covert security in [90] by proposing significant improvements for covert protocols based on TinyOT [46, 96, 131, 146]. These efficiency improvements, although presented for covert security, carry over to PVC security and FBC security.

1.2. Statement Oblivious Witness Encryption

One primitive that appears particularly challenging to instantiate from standard cryptographic assumptions is *witness encryption* [99]. Witness encryption, defined for some NP-complete² language \mathcal{L} , enables a party to encrypt a message m under a problem statement $x \in \mathcal{L}$ such that the ciphertext can only be decrypted by a party knowing a witness w to the problem statement x . Despite significant efforts and progress [34, 98, 99, 102, 105, 135], general-purpose witness encryption is still far from practical. The existing instantiations are based on strong assumptions

¹Concurrent to our work, [155] presented a transformation that follows a similar approach to ours. We discuss the differences and trade-offs between the two works in Chapter 3.

²It is widely assumed that a computationally bounded adversary is unable to solve a problem statement sampled from an NP-complete language. This is the subject of the famous $P \stackrel{?}{=} NP$ questions.

1. Introduction

and lack efficiency. Moreover, the standard notion of witness encryption does not consider the need to keep the statement used for encryption private. This rules out further interesting applications.

Goyal et al. [106] proposed a blockchain-based alternative to standard witness encryption. A recent line of work [27, 49, 101] has shown how blockchains can be exploited to nominate committees with a guaranteed minimal threshold of honest parties. Goyal et al. utilize this work to shift the trust assumption from strong cryptographic assumptions to trust in the committee provided by a blockchain. In [106], a ciphertext on a message m is created by secret sharing the plaintext among the committee members and tagging each share with the problem statement x . The committee members reveal their share of the message m to whoever provides a valid witness w for the tagged statement x . While this approach straightforwardly provides the functionality of witness encryption, it has the downside that the committee needs to store all currently active ciphertexts. Furthermore, in the protocol of [106], the statement used for encryption is inherently public. The authors do not consider the potential of hidden statements.

Hidden statement, on the one hand, can extend use cases of traditional witness encryption with additional privacy guarantees, such as time-lock encryption [135] with a hidden release time and dead-man’s switches [106] with hidden identities. On the other hand, they enable novel use cases – for example, in the decentralized finance sector, where witness encryption with hidden statements could be employed to execute transactions contingent on specific events, like a stock reaching a target price, without revealing the transaction or the event until it occurs.

Our Contribution. In [91], we adapt the committee-based approach to witness encryption but address both of the described shortcomings of [106]. In particular, we propose the notion of *statement oblivious threshold witness encryption (SO-TWE)* and provide instantiations based on different cryptographic building blocks. In SO-TWE, the committee members hold shares of an asymmetric decryption key and perform decryptions upon request. A client can use the corresponding public encryption key and a problem statement x to encrypt a message m locally. The resulting ciphertext c does not yield any information about x or m . For decryption, a client submits c to the committee along with a statement/witness-pair (x', w') . The committee members validate the pair and, if successful, reply with decryption shares computed based on the statement x' and their shares of the decryption key. Importantly, the committee members do not learn whether the statement submitted for decryption is the same as the one used for encryption, i.e., whether $x = x'$. Once the client receives enough decryption shares, it reconstructs a message m' . However, if the statement used for decryption is not the same as the

1. Introduction

one used for encryption, i.e., if $x' \neq x$, the reconstructed message is just a random string that leaks no information about the encrypted message m or statement x .

1.3. Virtual Trusted Third Party

A trusted third party can be seen as a service that is continuously available, strictly adheres to the expected behavior, keeps its state and communication secret, and is capable of performing complex computations. The existence of such a service would be of tremendous use and would render many cryptographic problems trivial. Unfortunately, there is no such thing as a trusted third party in the real world. However, there are systems employing cryptographic protocols or advanced hardware to offer trusted computations with some of the properties that constitute a trusted third party. In this thesis, we refer to such a service as *virtual trusted third party* (*V-TTP*).

There are three major approaches to virtual trusted third parties. First, a virtual trusted third party can be instantiated via a committee of servers that secret share the private state and perform the requested computations via secure multiparty computation (cf. among others [22, 31, 53, 121, 138]). This approach ensures the correct execution of the service and privacy of the service's state. However, multiparty computation adds a significant performance overhead to the requested computations, rendering it unsuitable for services that require complex computations. In addition, there is an inherent trade-off between the availability of the service and the privacy of the internal state; the higher the availability guarantees, the less malicious parties can be tolerated within the committee. A second approach is to rely on trusted hardware (cf. among others [17, 109, 129, 149]), usually referred to as *Trusted Execution Environments* (*TEE*). A TEE is a piece of hardware integrated into an operator's system that guarantees the correct execution of installed programs and secure storage of secrets. However, even though a TEE protects the service's secrets, guarantees correct execution, and is capable of conducting complex computations, the hardware is in possession of an operator that controls all incoming and outgoing communication. This allows the operator to censor communication such that the availability of a TEE-based virtual trusted third party cannot be guaranteed. Third, a virtual trusted third party can be realized in the form of a smart contract deployed to a public blockchain (cf. among others [30, 167, 173, 176, 177]). Smart contracts are highly available and guarantee the correct execution of the deployed code. However, traditional smart contracts require the contract state to be public and are unsuitable for complex computations.

1. Introduction

A long line of work (cf. among others [56, 73, 80, 120, 128, 164]) aims to address the limitations of traditional smart contract platforms, thereby enhancing the efficiency and privacy guarantees of smart contract-based V-TTPs. Many of those utilize TEEs or techniques known from MPC. However, existing solutions focus on either the privacy or efficiency of smart contracts or introduce additional shortcomings, such as the necessity for the parties involved in the execution of the smart contract to lock collateral; collateral can be withheld to punish unresponsive parties, thereby incentivizing responsiveness.

Our contribution. In [95], we revisit the problem of providing a virtual trusted third party based on smart contracts by proposing a novel smart contract system, POSE, built on top of traditional smart contract platforms such as Ethereum. POSE outsources the execution of smart contracts to a committee of TEE operators, one committee for each contract. This allows us to guarantee the privacy of the contract state and significantly increases the efficiency of the smart contract execution. At the same time, we avoid shortcomings of prior solutions proposing privacy- or efficiency-enhanced smart contract platforms, such as the need to lock collateral. However, we make a minor sacrifice regarding the liveness of the system compared to traditional smart contract platforms. While traditional smart contract platforms only require the underlying blockchain system to remain healthy, POSE requires, in addition to a healthy blockchain system, at least one TEE operator in a randomly selected pool to be honest, i.e., remain responsive.

1.4. Modern Programming Language for Decentralized Applications

Naturally, the utilization and optimization of blockchain features go hand in hand. During our work on blockchain-based cryptography, we have identified several shortcomings in the prevailing programming paradigm for *decentralized applications (dApps)*, programs composed of a smart contract deployed to a blockchain and clients interacting with the smart contract. Currently, clients and contracts are implemented as separate programs in different programming languages. Contracts exhibit a list of public functions that can be invoked by the clients. In order to protect the intended program flow against client-sided deviations, developers restrict the states, in which each function can be invoked, via manually added entry guards. This way, the developers implicitly encode the program flow as a finite state machine. While this programming model accurately reflects the functioning of the blockchain, in which contracts are invoked via transactions submitted by

1. Introduction

the clients, it makes the distributed program flow awkward to express and reason about. The difficulties in expressing the intended program flow can, in turn, increase the risk of implementation errors and mismatches in the client-contract interface. As contracts directly control financial assets, programming errors can result in significant financial losses [4, 5, 6].

Our contribution. In order to address the shortcomings of the prevailing approach to dApp development, we propose a novel programming language, **Prisma** [152]. **Prisma** aims to capture the internal semantics of dApps rather than reflecting the technical low-level processes of the smart contract platform. As the smart contract represents the central entity responsible for managing the global state of the dApp and deciding on the allowed state transition, we no longer model it as a passive entity invoked by clients. Instead, we interpret the smart contract as the active entity that is in charge of the control flow and passes it to the clients if input is required. This approach enables developers to explicitly specify the distributed program flow of the dApp with standard patterns akin to a *main* function known from other modern programming languages. Our compiler transforms **Prisma** code to finite state machine-style smart contract code that can be deployed to the blockchain and Scala client code that is executed by the clients. It automatically adds entry guards to every contract function, which protect the intended program flow against client-sided deviations. Furthermore, **Prisma** allows the implementation of both client and contract in a single unit with the same programming language, thus rendering mismatching communication impossible.

1.5. Thesis Outline

Chapter 2 presents definitions and building blocks used throughout this thesis and provides the necessary background on blockchains, smart contracts, and trusted execution environments. In Chapter 3, we detail the contribution of our publications [86, 88, 90] on covert security, publicly verifiable covert security, and financially backed covert security. Chapter 4 describes our publication [91] on statement oblivious witness encryption. In Chapter 5, we detail our work on the POSE protocol [95], which emulates a virtual trusted third party without the limitations known from prior approaches. Chapter 6 presents the contribution of our publication [152], where we propose the programming language **Prisma**, which adopts modern programming language concepts to the realm of decentralized applications. Chapter 7 concludes the thesis by discussing interesting directions for future work.

2. Preliminaries

This chapter introduces the notation used throughout this thesis and recalls relevant basics, definitions, and building blocks.

2.1. Notation and Convention

We denote the set $\{1, \dots, k\}$ by $[k]$, the statistical security parameter by κ , and the computational security parameter by λ . We use the abbreviation PPT to describe a probabilistic polynomial-time algorithm. We call a function $f : \mathbb{N} \rightarrow \mathbb{R}$ *negligible* in λ if for every positive integer c , there exists an integer λ' such that for every $\lambda > \lambda'$, it holds that $f(\lambda) < \frac{1}{\lambda^c}$. We denote a negligible function by $\text{negl}(\lambda)$. Let $\{\mathcal{X}_\lambda\}$ and $\{\mathcal{Y}_\lambda\}$ be two sequences of distributions, where both \mathcal{X}_λ and \mathcal{Y}_λ range over $\{0, 1\}^{l(\lambda)}$ for some $l(\lambda) = n^c$ with constant c . We call $\{\mathcal{X}_\lambda\}$ and $\{\mathcal{Y}_\lambda\}$ *computationally indistinguishable* if, for every PPT algorithm \mathcal{A} , it holds that $|\Pr[\mathcal{A}(1^\lambda, \mathcal{X}_\lambda) = 1] - \Pr[\mathcal{A}(1^\lambda, \mathcal{Y}_\lambda) = 1]| \leq \text{negl}(\lambda)$. We write $\{\mathcal{X}_\lambda\} \stackrel{c}{\equiv} \{\mathcal{Y}_\lambda\}$ to denote computational indistinguishability of $\{\mathcal{X}_\lambda\}$ and $\{\mathcal{Y}_\lambda\}$.

NP is the class of languages that can be decided by a deterministic Turing machine in polynomial time. More precisely, a language \mathcal{L} is in NP if there exists a relation $\mathcal{R}(x, w)$ computable in polynomial time in x such that for every problem statement $x \in \mathcal{L}$ there exists a witness w such that $\mathcal{R}(x, w) = \text{true}$ and for every $x \notin \mathcal{L}$ there exists no witness w such that $\mathcal{R}(x, w) = \text{true}$. A language \mathcal{L} in NP is called NP-complete if every algorithm \mathcal{A} deciding membership in \mathcal{L} in polynomial time can be used to decide membership in any other NP language in polynomial time. It is commonly assumed that there are no polynomial-time algorithms deciding NP-complete languages. This is the subject of the famous $P \stackrel{?}{=} NP$ question.

If not stated otherwise, we will adhere to the following conventions throughout this thesis. Adversaries are considered to be computationally bounded, i.e., they are modeled as PPT algorithms and always receive 1^λ as additional input. We neglect negligible attack probabilities of adversaries, e.g., if we state that something is impossible for a (PPT) adversary, it is impossible except with negligible probability. Further, we consider static corruption, i.e., require the adversary to select the parties that should be corrupted prior to the protocol execution.

2.2. (Publicly Verifiable) Covert Secure Multiparty Computation

The term *secure multiparty computation (MPC)* describes cryptographic protocols allowing a set of parties to compute a function f on their inputs such that nothing beyond the output is revealed. *Covert* security as introduced by Aumann and Lindell [14] is a security notion for multiparty computation and cryptographic protocols in general that allows the adversary to successfully cheat with a fixed probability $1 - \epsilon$, called the deterrence factor. However, the honest parties detect the cheating attempt with probability ϵ . *Publicly verifiable covert (PVC)* security as introduced by Asharov and Orlandi [12] is an extension to covert security that allows any honest party detecting a cheating attempt to prove the misbehavior of the detected party to any third party. In the following, we introduce the basic concepts of simulation-based security proofs, the prevalent technique to prove the security of cryptographic protocols, and provide formal definitions for covert secure and publicly verifiable covert secure multiparty computation.

Simulation-based security. The security of a cryptographic protocol is usually defined by specifying an ideal world that provides the same functionality as the cryptographic protocol is supposed to provide. In the ideal world, all parties communicate with an ideal trusted third party \mathcal{F} . In the case of MPC, the ideal-world execution has four simple steps: the honest parties send their input to \mathcal{F} , the ideal-world adversary sends the inputs of all corrupted parties on their behalf to \mathcal{F} , \mathcal{F} computes function f on the received inputs, and \mathcal{F} returns the output to the honest parties and the ideal-world adversary. It is easy to see that the output given to the parties is the result of the function evaluated on all inputs and that the adversary does not learn anything about the honest parties' inputs except what it can derive from the output. This is precisely what we require from an MPC protocol.

A protocol is secure if the ideal world and the real world are equivalent or, in other words, if all “attacks” that are possible in the real-world execution are also possible in the ideal-world execution. To prove the equivalence of the two worlds, cryptographers specify an ideal-world adversary, called *simulator*, that resides in the ideal world but generates a view for the real-world adversary that is indistinguishable from the adversary's view in a real-world execution. In order to generate a real-world adversarial view, the simulator internally executes the adversary and simulates a real-world execution to the adversary. In this simulated real-world execution, the adversary plays the role of the corrupted parties and

2. Preliminaries

the simulator the one of the honest parties. However, the simulator does not have direct access to the honest parties, e.g., their inputs, but only to the data it receives as ideal-world adversary from the trusted third party, e.g., the result of evaluating f . Since the simulator participates in the ideal-world execution, it also needs to translate the messages received by the adversary to messages it is supposed to send to the trusted party in the ideal-world execution, e.g., the inputs of the corrupted parties. This way, it translates attacks possible in the real world to attacks in the ideal world. As the adversary is executed as an internal subroutine of the simulator, the simulator can reset the adversary and restart the simulated real-world execution. This technique is called *rewinding*. Equivalence of the two worlds is shown by proving that the honest party's output and the view generated by the simulator in an ideal-world execution are computationally indistinguishable from the honest party's output and the view of the adversary in a real-world execution.

Often, it is not directly evident that the two tuples are indistinguishable. Therefore, equivalence is usually shown via a sequence of hybrid worlds gradually aligning the ideal-world execution to a real-world execution until a final hybrid identical to the real world. The cryptographer then proves indistinguishability between every two consecutive hybrid worlds, e.g., via reductions to the security guarantees of underlying cryptographic primitives. If the number of hybrids is polynomial in the security parameter, we can conclude from the indistinguishability between all consecutive hybrid worlds that the ideal world and the real world are computationally indistinguishable.

2.2.1. Covert Security

Aumann and Lindell [14] propose several notions of covert security. We focus on the one that is most prevalent in research, the *strong explicit cheat formulation (SECF)*. The definitions for covert security of Aumann and Lindell incorporate *identifiable abort*, i.e., require all honest parties to identify one halting party in case of an abort. In the two-party setting considered in [14], achieving identifiable abort is trivial and comes without any efficiency overhead. This is not the case in the multiparty setting. As we consider identifiable abort an independent research area, we remove this property from the notion of covert security considered in this thesis. In the following, we provide a formal definition of the SECF without identifiable abort. We take the definition almost verbatim from [86].

In the *real world*, the parties jointly compute the desired function f using a protocol π . Let n be the number of parties and let $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$, where $f = (f_1, \dots, f_n)$, be the function realized by π . We define for every input vector $\bar{x} = (x_1, \dots, x_n)$ the output vector $\bar{y} = (f_1(\bar{x}), \dots, f_n(\bar{x}))$ where party P_i

2. Preliminaries

with input x_i obtains the output $f_i(\bar{x})$. During the execution of π , the adversary \mathcal{A} can corrupt a subset $\mathcal{I} \subset [n]$ of all parties. We define $\text{REAL}_{\pi, \mathcal{A}(z), \mathcal{I}}(\bar{x}, 1^\lambda)$ to be the output of the protocol execution π on input $\bar{x} = (x_1, \dots, x_n)$ and security parameter λ , where \mathcal{A} on auxiliary input z corrupts parties \mathcal{I} . We further specify $\text{OUTPUT}_i(\text{REAL}_{\pi, \mathcal{A}(z), \mathcal{I}}(\bar{x}, 1^\lambda))$ to be the output of party P_i for $i \in [n]$.

In contrast, in the *ideal world*, the parties send their inputs to a trusted party \mathcal{F} which computes function f and returns the result. Hence, the computation in the ideal world is correct by definition. The security of π is analyzed by comparing the ideal-world execution with the real-world execution. The ideal world in covert security is slightly adapted in comparison to the standard secure computation model. In covert security, the ideal world allows the adversary to cheat and cheating is detected with some fixed probability ϵ , which is called the *deterrence factor*. Let $\epsilon : \mathbb{N} \rightarrow [0, 1]$ be a function; the execution in the ideal world is defined as follows.

Inputs: Each party obtains an input; the i -th party's input is denoted by x_i . We assume that all inputs are of the same length. The ideal-world adversary \mathcal{S} receives an auxiliary input z .

Send inputs to trusted party: Any honest party P_j sends its received input x_j to the trusted party. The corrupted parties, controlled by \mathcal{S} , may either send their received input or send some other input of the same length to the trusted party. This decision is made by \mathcal{S} and may depend on the values x_i for $i \in \mathcal{I}$ and auxiliary input z . Denote the vector of inputs sent to the trusted party by \bar{w} .

Abort options: If a corrupted party sends $w_i = \text{abort}$ to the trusted party as its input, then the trusted party sends **abort** to all of the honest parties and halts. If a corrupted party sends $w_i = \text{corrupted}_i$ to the trusted party as its input, then the trusted party sends **corrupted_i** to all of the honest parties and halts. If multiple parties send **abort** (resp., **corrupted_i**), then the trusted party reacts only to one of them (say, the one with the smallest i). If both **corrupted_i** and **abort** messages are sent, then the trusted party ignores the **corrupted_i** message.

Attempted cheat option: If a corrupted party sends $w_i = \text{cheat}_i$ to the trusted party as its input, then the trusted party works as follows:

1. With probability ϵ , the trusted party sends **corrupted_i** to the adversary and all of the honest parties.
2. With probability $1 - \epsilon$, the trusted party sends **undetected** to the adversary along with the honest parties' inputs $\{x_j\}_{j \notin \mathcal{I}}$. Following this, the adversary sends the trusted party output values $\{y_j\}_{j \notin \mathcal{I}}$ of its choice for the honest parties. Then, for every $j \notin \mathcal{I}$, the trusted party sends y_j to P_j .

2. Preliminaries

The ideal execution then ends at this point. If no w_i equals `abort`, `corruptedi` or `cheati`, the ideal execution continues below.

Trusted party answers adversary: The trusted party computes $(y_1, \dots, y_n) = f(\bar{w})$ and sends y_i to \mathcal{S} for all $i \in I$.

Trusted party answers honest parties: After receiving its outputs, the adversary sends either `abort` or `continue` to the trusted party. If the trusted party receives `continue` then it sends y_j to all honest parties P_j ($j \notin \mathcal{I}$). Otherwise, if it receives `abort`, it sends `abort` to all honest parties.

Outputs: An honest party always outputs the message it obtained from the trusted party. The corrupted parties output nothing. The adversary \mathcal{S} outputs any arbitrary (PPT computable) function of the initial inputs $\{x_i\}_{i \in \mathcal{I}}$, the auxiliary input z , and the messages obtained from the trusted party.

We denote by $\text{IDEALC}_{f, \mathcal{S}(z), \mathcal{I}}^\epsilon(\bar{x}, 1^\lambda)$ the output of the honest parties and the adversary in the execution of the ideal model as defined above, where \bar{x} is the input vector and the adversary \mathcal{S} runs on auxiliary input z .

Definition 2.1 (Covert security with ϵ -deterrent). Let f, π , and ϵ be as above. Protocol π is said to *securely compute f in the presence of covert adversaries with ϵ -deterrent* if for every non-uniform PPT adversary \mathcal{A} for the real model, there exists a non-uniform PPT adversary \mathcal{S} for the ideal model such that for every $\mathcal{I} \subseteq [n]$, every balanced vector $\bar{x} \in (\{0, 1\}^*)^n$, and every auxiliary input $z \in \{0, 1\}^*$:

$$\{\text{IDEALC}_{f, \mathcal{S}(z), \mathcal{I}}^\epsilon(\bar{x}, 1^\lambda)\}_{\lambda \in \mathbb{N}} \stackrel{c}{\equiv} \{\text{REAL}_{\pi, \mathcal{A}(z), \mathcal{I}}(\bar{x}, 1^\lambda)\}_{\lambda \in \mathbb{N}}$$

In order to prevent detecting parties that behave honestly except that they might abort as corrupted, the definition of *non-halting detection accurate* was introduced in [14]. The definition uses the notion of a *fail-stop* party, which acts honestly, except that it may halt early.

Definition 2.2. A protocol π is *non-halting detection accurate* if for every honest party P_j and every honest or fail-stop party P_k the probability that P_j outputs `corruptedk` is negligible.

2.2.2. Publicly Verifiable Covert Security

While the original notion of PVC security [12] was restricted to two parties, we [88] have extended the notion to the multiparty setting. In the following, we provide our formal definition of PVC security in the multiparty setting. The definition is taken almost verbatim from [88] (with some minor modifications).

2. Preliminaries

In addition to the covert secure protocol π , PVC security defines two algorithms, **Blame** and **Judge**. **Blame** takes as input the view of an honest party P_i after P_i outputs `corruptedj` in the protocol execution for $j \in I$ and returns a certificate `cert`, i.e., $\text{cert} := \text{Blame}(\text{view}_i)$. The **Judge**-algorithm takes as input a certificate `cert` and outputs the identity `idj` if the certificate is valid and states that party P_j behaved maliciously; otherwise, it returns `none` to indicate that the certificate was invalid.

Moreover, we require that the protocol π is slightly adapted such that an honest party P_i computes $\text{cert} = \text{Blame}(\text{view}_i)$ and broadcasts `cert` after cheating has been detected. We denote the modified protocol by π' . Notice that due to this change, the adversary gets access to the certificate. By requiring simulatability, it is guaranteed that the certificate does not reveal any private information.

Definition 2.3 (Covert security with ϵ -deterrent and public verifiability). Let f, π', Blame , and **Judge** be as above. The triple $(\pi', \text{Blame}, \text{Judge})$ *securely computes f in the presence of covert adversaries with ϵ -deterrent and public verifiability* if the following conditions hold:

1. (**Simulatability**) The protocol π' securely computes f in the presence of covert adversaries with ϵ -deterrent according to the strong explicit cheat formulation (see Definition 2.1) and non-halting detection accurate (see Definition 2.2).

2. (**Accountability**) For every PPT adversary \mathcal{A} corrupting parties P_i for $i \in \mathcal{I} \subset [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0, 1\}^*)^{n+1}$ the following holds:

If $\text{OUTPUT}_j(\text{REAL}_{\pi', \mathcal{A}(z), \mathcal{I}}(\bar{x}, 1^\lambda)) = \text{corrupted}_i$ for $j \in [n] \setminus \mathcal{I}$ and $i \in \mathcal{I}$ then:

$$\Pr[\text{Judge}(\text{cert}) = \text{id}_i] > 1 - \text{negl}(\lambda),$$

where `cert` is the output certificate of the honest party P_j in the execution.

3. (**Defamation Freeness**) For every PPT adversary \mathcal{A} corrupting parties P_i for $i \in \mathcal{I} \subset [n]$ and interacting with the honest parties, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0, 1\}^*)^{n+1}$ and all $j \in [n] \setminus \mathcal{I}$:

$$\Pr[\text{cert}^* \leftarrow \mathcal{A}; \text{Judge}(\text{cert}^*) = \text{id}_j] < \text{negl}(\lambda).$$

2.3. Cryptographic Building Blocks

This section introduces the cryptographic building blocks utilized throughout this thesis. As we focus on providing an overview of our contributions, we restrict the description of the building blocks to high-level intuitions.

Hash functions. A *cryptographic hash function* H is a function mapping an input of arbitrary length to a fixed-length output. While inputs and outputs are usually encoded as bit strings, it is also possible to interpret them as elements of specific algebraic structures. Security-wise, cryptographic hash functions are required to satisfy *collision resistance* stating that it is not possible for any PPT adversary to find two different input strings x_1 and x_2 such that $H(x_1) = H(x_2)$.

Random oracle. A *random oracle* is a theoretical cryptographic concept providing a trusted service that is available to all parties and responds to every unique input with a truly random element chosen from its output domain. The oracle stores all answered queries such that it can produce the same output for repeated queries with the same input. While it is not possible to instantiate a random oracle in a provable secure way, the oracle is usually instantiated with a cryptographic hash function in real-world use cases. The random oracle is, therefore, often seen as an idealized hash function.

Bilinear pairings. A *bilinear pairing* is composed of three cyclic groups \mathbb{G}_1 , \mathbb{G}_2 , and \mathbb{G}_T of prime order p , generators $g_1 \in \mathbb{G}_1$, $g_2 \in \mathbb{G}_2$, and an efficiently computable pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. It needs to hold that $e(u^a, v^b) = e(u, v)^{ab}$ for all $(u, v, a, b) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{Z}_p \times \mathbb{Z}_p$. We assume the *decision bilinear Diffie-Hellman (DBDH)* assumption to hold in bilinear pairings. The DBDH states that it is impossible for any PPT adversary receiving a group description $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e)$ and a challenge tuple (g^a, g^b, R) to distinguish whether R is sampled uniformly random from \mathbb{G}_T or defined as $R = e(g_1, g_2)^{ab}$.

(Homomorphic) secret sharing. A *t-out-of-n secret sharing scheme* allows a party to split a secret s into n shares $\{s_i\}_{i \in [n]}$ such that any subset of $t' < t$ shares does not leak any information about the secret while any subset of t distinct shares is sufficient to reconstruct the secret s . If $t = n$, a simple secret sharing scheme can be built by encoding s as bit-string and sampling all s_i uniformly random under the constraint that $s = \bigoplus_{i \in [n]} s_i$. The most commonly used secret sharing scheme for $t < n$ is the Shamir secret sharing [157], in which the secret s is shared by sampling a uniform random degree $t - 1$ polynomial $p(X)$ with $p(0) = s$ and

2. Preliminaries

defining $s_i = p(i)$ for $i \in [n]$. Given t distinct evaluations of the polynomial $p(X)$, it is possible to interpolate $s = p(0)$ using LaGrange interpolation.

For some use cases, it is desirable for the secret sharing scheme to exhibit homomorphic properties, i.e., to enable parties to locally manipulate their shares of the secret s such that the reconstruction applied to the manipulated shares yields a value $F(s)$ for a known function F . The Shamir secret sharing, for example, is homomorphic under addition. Given a sharing of a secret x and a sharing of a secret y , both with the same parameter t , parties can compute a sharing of $z = x + y$ by adding their local shares, i.e., $z_i = x_i + y_i$. In case the secret sharing scheme allows the evaluation of arbitrary functions on the shared secret, we call the scheme *homomorphic secret sharing (HSS)* [40, 41].

Encryption. An *encryption scheme* allows a party to encrypt a message m with an encryption key k_e such that the resulting ciphertext c does not leak any information about the message m except to the parties holding the decryption key k_d . We call an encryption scheme *symmetric* if the key used for encryption is the same as the key used for decryption, i.e., if $k_e = k_d$. Otherwise, we call the scheme *asymmetric*. In the context of *asymmetric* encryption, we call the encryption key *public key* and the decryption key *secret key*. While symmetric encryption schemes are usually more efficient, asymmetric schemes have the advantage that the encryption key can be announced publicly, such that the party holding the corresponding secret key can receive encrypted messages from arbitrary senders without having to establish a shared secret key.

Encryption schemes can provide different levels of security, most importantly *indistinguishably under chosen plaintext attacks (IND-CPA)* and *indistinguishably under chosen ciphertext attacks (IND-CCA)*. Both notions state that a PPT adversary receiving a ciphertext, which is the result of encrypting one of two messages, both selected by the adversary, is not able to differentiate which of the two messages has been encrypted. In IND-CPA security, the adversary is given the power to query additional ciphertexts for arbitrary messages. In IND-CCA security, the adversary is additionally given the power to query decryptions of arbitrary ciphertexts, just not for the ciphertext that should be distinguished.

Digital signatures. A *digital signature scheme* is a cryptographic tool to ensure the authenticity and integrity of digital messages and documents. It allows a holder of a secret signing key sk , called signer, to generate a signature σ on a message m such that everyone in possession of the public verification key vk corresponding to sk can verify that the message m originated from the signer and has not been

2. Preliminaries

modified during transmission. A signature scheme is required to satisfy *correctness* and *unforgeability*. Correctness states that for any given key pair $(\mathbf{sk}, \mathbf{vk})$, any signature σ created on a message m with secret key \mathbf{sk} verifies successfully under verification key \mathbf{vk} with respect to message m . Unforgeability states that for any given key pair $(\mathbf{sk}, \mathbf{vk})$, it is impossible for a PPT adversary receiving only \mathbf{vk} to find a signature/message-pair that verifies successfully under \mathbf{vk} .

There are a variety of notions of digital signature schemes providing different levels of security or additional functionalities. In this thesis, we consider *one-time signatures* and signatures that are *existentially unforgeable under chosen-message attacks*. In one-time signature schemes, each signing key can only be used once. Unforgeability cannot be guaranteed if a signing key is used more than once. Existential unforgeability under chosen-message attacks guarantees unforgeability even if the adversary is able to query signatures under \mathbf{vk} for arbitrary messages. Naturally, a signature for a message that has already been queried is not considered a forgery.

Commitments. A *commitment scheme* enables a party, Alice, to compute a commitment/opening-pair (c, d) for a message m . While the value c already commits Alice to message m , it does not leak any information about m . To open c to m , Alice has to publish d . However, Alice cannot open c to any other message but m . A commitment scheme is called *correct* if a commitment/opening-pair computed for a message m always opens to m . From a security standpoint, we require the commitment scheme to be *hiding* and *binding*. The former states that an adversary selecting two messages and receiving a commitment for either of the two cannot distinguish which message the challenger has committed to. The latter states that an adversary is unable to find a commitment c and two openings that open c to two different valid messages. Both properties can be defined with respect to both computationally bounded (PPT) and unbounded adversaries.

Merkle Trees. A *Merkle tree* [141] is a cryptographic data structure that identifies a large list L by a single hash r and allows the efficient and secure verification of membership in L . The data structure is defined as a binary tree whose leaves are labeled with the hashes of the elements in L . Each intermediate node is labeled with a hash computed on the concatenation of the labels of its two child nodes. The label of the tree root is the identifier r . Due to the collision resistance of the hash functions, it is computationally infeasible to construct two distinct trees with the same root and, hence, with the same identification hash. Therefore, it is safe to assume that each list has a unique identifier. In order to prove that a particular

2. Preliminaries

element e is at the k -th position of the list identified by r , a prover provides the element e , the position k , and the labels of the sibling nodes required to recompute all labels on the path from e to the root r . We call such a proof a *Merkle proof*

(Threshold) identity-based encryption. *Identity-based encryption (IBE)* [33, 158] enables parties to encrypt messages to a specific identity, e.g., an e-mail address, instead of encrypting the message to a public key. An IBE scheme consists of four algorithms: a setup, an identity key generation, an encryption, and a decryption. The setup algorithm generates public parameters and a master key. The identity key generation algorithm takes the public parameters, the master key, and an identity as input and generates an identity key. The encryption algorithm takes the public parameters, an identity, and a message as input and generates a ciphertext. The decryption algorithm takes the public parameters, an identity key, and a ciphertext as input and returns a plaintext.

In this thesis, we require *correctness* and *security under chosen identity attacks*. Correctness states that a ciphertext created under a particular identity can successfully be decrypted with the corresponding identity key. Security under chosen identity attacks requires that it is not possible for any PPT adversary, that selects two messages m_0 and m_1 as well as an identity id^* and receives a ciphertext of either m_0 or m_1 under id^* , to distinguish which message has been encrypted. This needs to hold even if the adversary is able to query arbitrary identity keys, just not the one for identity id^* .

In order to mitigate the single point of failure represented by the holder of the master key, the scheme can be thresholdized. This means that there is a committee of servers holding shares of the master key such that an identity key can only be generated by t collaborating servers. We call such a scheme *threshold identity-based encryption* [16]. Some use cases require the identity under which a message has been encrypted to be private. If an IBE scheme ensures that a ciphertext does not leak any information about the identity used for encryption, we call the scheme *anonymous identity-based encryption* [35, 100]. Finally, we call a scheme that combines the thresholdization with the anonymity property *anonymous threshold identity-based encryption (A-TIBE)* [88].

(Threshold) tag-based encryption. A *tag-based encryption scheme (TBE)* [139] is an asymmetric encryption scheme in which ciphertexts are created with respect to a specific tag. The decryption of a ciphertext is only successful if it is executed with the same tag that was used for encryption. The decryption in TBE schemes can be thresholdized such that the decryption key is secret shared among

2. Preliminaries

a committee of n servers and decryption can only be performed by t collaborating servers. Such a thresholdized scheme is called a *threshold tag-based encryption scheme (TTBE) scheme* [9]. Security of both TBE and TTBE is defined analogously to standard asymmetric encryption.

Zero-knowledge proofs. A *zero-knowledge proof system* is a cryptographic tool that allows a prover to prove to a verifier that a particular statement is true without revealing any information beyond the fact that the statement is true. There are efficient zero-knowledge proof systems for many cryptographically interesting statements, e.g., to prove knowledge of the discrete logarithm of a given group element.

Oblivious Transfer. A *s-out-of-n oblivious transfer* is a two-party protocol executed between a sender and a receiver, in which the sender transfers t out of its n messages to the receiver without learning which of the messages has been transferred. The sender's input is a list of n messages (m_1, \dots, m_n) . The receiver's input is a set of indices \mathcal{S} of size s . As a result of the protocol, the receiver learns messages $(m_{\mathcal{S}[1]}, \dots, m_{\mathcal{S}[s]})$. The protocol guarantees that a PPT sender does not learn any information about \mathcal{S} and a PPT receiver does not learn any information about any message $m_i : i \notin \mathcal{S}$.

Time-lock puzzles. *Time-lock puzzles (TLP)*, initially introduced by [154], allow a party to lock a message m into a puzzle p such that no PPT adversary is able to extract m in time smaller than \mathcal{T} . Simultaneously, the puzzle guarantees that every (reasonably equipped) honest party does not need much longer than \mathcal{T} to extract the message. We call the process of extracting the message *solving*. In [88], we extend the notion of time-lock puzzles with a *verifiability* property that allows a party that has solved a puzzle p to generate a proof π which can be used by any third party to compute the message m from puzzle p efficiently, i.e., in time much smaller than \mathcal{T} . We call the process of extracting the message based on a proof *opening*. The verifiability property can be integrated into TLP schemes based on [154] by using techniques known from verifiable delay functions [150, 172].

2.4. Blockchain and Smart Contracts

The term *blockchain* describes a distributed append-only database known as the *ledger*, which is maintained by a decentralized peer-to-peer network of nodes. Within this network, each node maintains its own copy of the ledger. The ledger

2. Preliminaries

comprises a sequence of blocks, where each block includes the hash of its predecessor. A block consists of a header, containing metadata such as timestamps or the hash of the previous block, and a body, which encompasses a list of transactions. The body is linked to the header through the inclusion of the root of a Merkle tree computed over the transaction list. This method of linking blocks with each other and with their transactions ensures the fixed order of both blocks and transactions within them. Nodes within the network execute a consensus protocol that ensures that all nodes share the same copy of the ledger and guarantees the immutability of blocks once appended to the blockchain.

In blockchain-based cryptocurrencies, transactions represent transfers of the system's currency. Coins are owned by public signature keys, representing identities, and are transferred through transactions signed with the corresponding private keys. Transactions are executed sequentially in accordance with the predetermined order defined by the blockchain. Consequently, the state of the blockchain, i.e., the balances of all parties, after appending a specific block is unambiguously defined by the list of transactions included in the blockchain up to that block.

Special nodes, called *proposers*, are responsible for gathering newly submitted transactions, aggregating them into blocks, and appending them to the blockchain. Proposers receive compensation for their efforts in the form of fees included in transactions and the issuance of new currency per block, both of which are directly allocated to the block proposer upon the block's acceptance into the blockchain. To prevent the inclusion of maliciously generated blocks, all nodes in the network validate received blocks and discard invalid ones. For instance, a block is considered invalid if one of its transactions transfers coins from party A to B without being signed by A or spends more coins than A possesses.

Blockchain consensus. The major purpose of the consensus protocol is to select the blocks to be appended to the blockchain. The consensus protocol usually operates under the assumption of an honest majority such that consensus and immutability of appended blocks are assured only if the majority of the network participants behave honestly. However, defining the majority in a decentralized network is challenging as an adversary can efficiently create fake identities, which is known as *sybil attack*. Consequently, blockchain consensus protocols deploy more sophisticated approaches relying on proxies to determine the majority. The two major categories of blockchain consensus protocols are *proof-of-work (PoW)* and *proof-of-stake (PoS)*.

PoW, introduced by Bitcoin and initially adopted by Ethereum [173], represents the original approach to blockchain consensus. PoW leverages the computational power within the system as a proxy for majority determination, thereby guaran-

2. Preliminaries

teeing consensus and immutability if at least half of the computational power is honest. In PoW, proposers creating new blocks have to solve a computationally intensive cryptographic puzzle in order to append their block to the blockchain. For instance, in Bitcoin, proposers have to incorporate a nonce into the block header such that the hash of the header commences with a specified number of zeros, known as the difficulty. The difficulty automatically adjusts to sustain a consistent block creation rate. Upon publication of a new valid block, other proposers stop solving their own puzzle, create a new block extending the newly appended one, and restart the puzzle-solving process. In cases where two proposers simultaneously propose new blocks, a *fork* occurs, necessitating other proposers to choose which chain to extend. Eventually, one of the competing chains will outpace the other, establishing itself as the valid chain and invalidating the forked chain. Consequently, PoW-based blockchains consider transactions included in a block as final and immutable only after confirmation by a certain number of successors.

In PoS, proposers receive voting power according to the amount of currency they own in the system. PoS has been adopted by most of the more recent blockchains, such as [2, 15, 29, 52, 160, 166]. Also, Ethereum [173] recently shifted to PoS. While all consensus protocols in the PoW category follow the same blueprint explained above, the design space of PoS protocols is much broader. The only common denominator among PoS protocols is the random assignment of the block creation to proposers based on the currency they possess in the system.

Smart contracts. Some blockchains, such as [2, 15, 29, 52, 83, 160, 166], support the deployment and execution of Turing-complete programs, so-called *smart contracts*, on the blockchain. In order to deploy a smart contract, a user submits a special transaction containing the contract’s code. The code exhibits a list of functions that can be invoked by users. Users execute contracts by submitting transactions to the network specifying the contract, the function that should be executed, and the input to that function. Smart contracts are capable of receiving and transferring coins according to their program logic. The blockchain state maintained by each node is no longer a list of balances but also includes the smart contracts deployed to the blockchain, defined by their code and state. When creating a new block, a proposer computes the updated blockchain state by sequentially executing all transactions within the block. Subsequently, the proposer aggregates this state into a tree structure and includes the root in the block header.

The blockchain’s consensus protocol ensures the correct execution of deployed contracts. All nodes need to recompute the transactions of received blocks. They then compare the resulting blockchain state with the state root included in the block header. If a proposer includes an incorrect root in the block header, the

2. Preliminaries

block is rejected, and the transactions within are reversed. However, to validate the correct execution of contracts and, hence, the blockchain state, it's crucial that the state after appending a block is unequivocally defined by the list of all transactions included in the blockchain up to that block. Thus, the state of each smart contract must be public, contract computations must be deterministic, and the execution order must be fixed.

Given that transactions now involve complex computations beyond simple monetary transfers, smart contract-capable blockchains employ a more sophisticated mechanism for determining transaction fees. Transaction complexity is captured via an internal metric called gas. Each operation within the execution environment consumes a specific amount of gas. Notably, permanent storage allocation is among the costliest operations, emphasizing the importance of optimizing smart contract storage consumption. Users specify a maximum gas limit and the price they pay per consumed gas when submitting a transaction. Once the transaction is included into the blockchain, fees are paid to the proposer based on the actual gas consumption. If a transaction runs out of gas or the user runs out of funds before the transaction's computation is finished, the transaction is reverted. However, the fees are still paid as the proposer and the nodes still need to compute the transaction in order to determine the transaction failure. To prevent excessive computations within a single block, a maximum gas limit is imposed. While this mechanism ensures liveness and prevents issues like infinite loops, it also imposes constraints on the complexity of smart contracts. Due to the limited complexity, the computations of smart contracts are highly expensive. At the time of writing, a simple monetary transfer in Ethereum costs around \$0.7 USD [84].

2.5. Trusted Execution Environments

We use the term *trusted execution environment (TEE)* to describe an isolated execution environment integrated into a processor that guarantees confidentiality and integrity of its data and processes. Simply put, TEEs ensure the correct execution of their programs and protect their data against external access. TEEs rely on specialized hardware that is integrated into a host system. We call the party possessing the host system the TEE *operator*. Operators can install programs called *enclaves* on their TEE. TEEs typically provide an attestation mechanism allowing operators to prove to an external party that an enclave is running a specific program. When necessary, the attestation report can be augmented with additional data regarding the TEE, e.g., the enclave's public keys.

Even though the TEE protects its programs and data, it remains under the

2. Preliminaries

physical control of a potentially malicious operator. Consequently, it depends on the operator for all communication with the outside world. While enclaves can protect the integrity and confidentiality of their communication by setting up encryption and authentication keys with their communication partners, an operator can still drop and replay messages. Operators can further shut down their TEE and, hence, the services provided by the enclaves running on their TEE. These limitations of TEEs need to be considered when deploying them in cryptographic protocols.

The two most commonly used examples of TEEs are Intel SGX [62, 113, 140] and ARM TrustZone [10]. We stress that existing TEE systems are far from being flawless. There is a continuous line of work showing attacks on existing TEE systems, e.g., [28, 43, 45, 50, 134], and proposing countermeasures, e.g., [1, 18, 42, 63, 159]. However, we assess research on improving TEE security orthogonal to our work. In this thesis, we treat TEEs as a general concept independent of real-world instantiations and assume them to provide the intended security guarantees.

3. Financially-backed Covert Security¹

Secure multiparty computation (MPC) allows a set of n parties to jointly compute a function f on their inputs such that nothing beyond the output of that function is revealed. While originally MPC was mainly studied by the cryptographic theory community, in recent years, many industry applications have been envisioned in areas such as machine learning [125], databases [169], blockchains [181] and more [8, 144]. One of the main challenges for using MPC protocols in practice is their huge overhead in terms of efficiency.

One popular approach to deal with the efficiency overhead is to split protocols into an input-independent offline and an input-dependent online phase. The offline protocol carries out precomputations that are utilized to speed up the online protocol, which securely evaluates the desired function. The main idea of this approach is that the offline protocol can be executed continuously *in the background* and the online protocol is executed ad-hoc once input data becomes available or output data is required. Since the performance requirements for the online protocol are usually much stricter, the offline part should cover the most expensive protocol steps. Examples of such offline protocols are the circuit generation of garbling schemes as in authenticated garbling [76, 122, 171, 178] or the generation of correlated randomness in the form of Beaver triples [23] in secret sharing-based protocols such as in SPDZ [69, 71] or the TinyOT-family [46, 96, 131, 146].

Another aspect with a strong impact on the protocol efficiency is the adversarial model. The two standard adversarial models of MPC are *semi-honest* and *malicious* security. While semi-honest adversaries follow the protocol description but try to derive information beyond the output from the interaction, malicious adversaries can behave in an arbitrary way. MPC protocols in the malicious adversary model provide stronger security guarantees at the cost of significantly less efficiency [76, 123]. As a middle ground between good efficiency and high security, Aumann and Lindell [14] introduced the notion of *security against covert*

¹Parts of the introductory section have been taken verbatim from [85, 87, 89] with some minor adjustments.

3. Financially Backed Covert Security

adversaries. As in the malicious adversary model, corrupted parties may deviate arbitrarily from the protocol specification, but the protocol ensures that cheating is detected with a fixed probability ϵ , called *deterrence factor*. The rationale behind covert security is that the reputational damage connected to a detected cheating attempt deters adversaries from cheating.

Although cheating can be detected in covert security, a party participating in the protocol cannot transfer knowledge about malicious behavior to other (external) parties. This shortcoming was addressed by Asharov and Orlandi [12] with the notion of *covert security with public verifiability* (PVC). Informally, PVC enables honest parties to create a publicly verifiable certificate about the detected malicious behavior, called *proof of misbehavior* (*PoM*). This certificate can subsequently be checked by any other party (usually referred to as *judge*), even if this party did not contribute to the protocol execution. PVC secure protocols for the two-party case were presented by [12, 116, 126, 182]. Further, Damgård et al. [70] showed a generic compiler from semi-honest to PVC security for the two-party setting and gave an intuition on how to extend their compiler to the multiparty case.

While PVC security seems to solve the major shortcoming of covert security at first glance, in many settings, PVC security is still not sufficient; especially when only the digital identities of the parties are known, e.g., on the Internet. In such a setting, a real party can create a new identity without suffering from a damaged reputation in the sequel. Hence, malicious behavior needs to be punished in a different way. The obvious choice is to punish malicious parties financially, as already suggested by [182]. While it is possible to realize financial punishments via the legal system, this would necessitate additional trust assumptions and a cumbersome and expensive setup. A more convenient approach is to utilize a smart contract. Smart contracts are highly available, capable of directly handling financial assets, and reliably perform exactly the specified computation. A naive approach towards realizing financial punishments via a smart contract is to require all parties to lock a security deposit in a smart contract before running a PVC secure protocol. Upon detection of a cheating attempt, a party submits the PoM obtained by the protocol to the contract, which retains the deposit of the malicious party and refunds the others. If the protocol ends without malicious behavior, the contract refunds all parties. Unfortunately, this approach is not practical. Since every instruction executed by a smart contract costs fees, it is highly important to keep the amount of computation performed by a contract small. However, verification of PoMs in existing PVC secure protocols is performed in a naive way that requires the judge to recompute a whole protocol execution. In [182], the authors address this problem by augmenting a PVC protocol with a highly efficient interactive judging procedure, which allows them to efficiently instantiate

3. Financially Backed Covert Security

the judge via a smart contract. Unfortunately, their work is tailored to a specific two-party PVC secure protocol based on garbled circuits. In this thesis, we aim to generalize the results of [182] by showing how to upgrade arbitrary semi-honest secure protocols to covert security with efficient financial punishment. However, before going into the details of our work, we first provide the necessary background on covert secure and PVC secure protocols, which serves as the basis for our work.

Covert security from cut-and-choose. All previously proposed covert secure protocols amplify the security of a semi-honest base protocol by applying the cut-and-choose technique. In the offline/online setting, this works as follows: Parties first commit to t random seeds. Next, they execute t parallel instances of a semi-honest secure protocol, implementing the offline phase. In the i -th instance, each party derives all of its randomness from its i -th committed seed. This makes the protocol instance deterministic and, hence, auditable. After execution of the semi-honest instances, parties agree for each instance on the instance transcript, i.e., the hashes of all messages sent during the execution of the instance. In the following, we will call such an agreement on message hashes a *public transcript of message hashes*. If the semi-honest protocol relies on private party-to-party communication, security is preserved by integrating a key exchange, e.g., Diffie-Hellman [75], into the protocol and encrypting all messages with the exchanged keys. After agreeing on the transcripts, the parties jointly select s random instances for auditing, in the following denoted as *watchlist*. The parameter s is usually set to $t-1$. As all parties share the same watchlist, we call the watchlist *global*. For each of the selected instances, all parties open their committed random seeds. Parties that refuse to provide correct openings are considered cheaters. If all commitments are opened successfully, each party locally emulates the audited instances and compares the computed messages to the agreed transcripts. In case of a mismatch, the first party that deviated from the protocol description is considered to be the cheater. It is important to identify the first cheating attempt, as an incorrect message can be the result of either malicious behavior or another incorrect message received in a previous round. If there has not been any cheating attempt, parties utilize the output of one of the remaining unchecked instances as the output of the offline phase. Since the audited instances are chosen uniformly at random, any cheating attempt is detected with probability at least $\frac{s}{t}$. After successfully completing the covert secure offline phase, the parties run a maliciously secure online phase. Therefore, it is necessary that the instances of the offline protocol, even though they are just semi-honestly secure, provide precomputation material of the form that is required by the maliciously secure online phase.

Auditing in the above blueprint requires the inputs of all parties to the audited

3. Financially Backed Covert Security

instances. While this approach works fine for input-independent offline protocols, in which the inputs are just random seeds, it becomes problematic in the input-dependent setting, where parties have actual private inputs. For input-dependent protocols, the cut-and-choose technique is, therefore, applied in a different way following the party virtualization paradigm introduced by [119]. Let $f(x_1, \dots, x_n) = (y_1, \dots, y_n)$ be the function that is to be computed and

$$f'(x_1^1, \dots, x_1^t, \dots, x_n^1, \dots, x_n^t) = (y_1^1, \dots, y_1^t, \dots, y_n^1, \dots, y_n^t) \text{ such that}$$

$$f\left(\bigoplus_{j \in [t]} x_1^j, \dots, \bigoplus_{j \in [t]} x_n^j\right) = \left(\bigoplus_{j \in [t]} y_1^j, \dots, \bigoplus_{j \in [t]} y_n^j\right).$$

In order to compute f with covert security, the parties run a semi-honest secure $(t \cdot n)$ -party protocol evaluating function f' , in which each real party P_i with input x_i simulates t virtual parties P_i^1, \dots, P_i^t with inputs x_i^1, \dots, x_i^t such that $x_i = \bigoplus_{j \in [t]} x_i^j$. All virtual parties derive all their randomness from precommitted random seeds. For auditing, each party discloses the inputs and seeds of s of its virtual parties such that their messages can be recomputed by the other parties. This again leads to a covert protocol with a deterrence factor of at least $\frac{s}{t}$. However, as it is not possible to recompute all of the incoming messages of the audited parties, which is required for auditing, it is necessary for the agreed instance transcript to contain all messages in full (instead of just a hash). In the following, we will call such a transcript a *public transcript of messages*.

PVC security from cut-and-choose. PVC secure protocols also follow the cut-and-choose technique but require some modifications to the approach explained above. In the following, we will focus on input-independent protocols. However, the explained problems and techniques translate to input-dependent protocols in a straightforward way. The naive approach for achieving PVC security from covert security is to require the parties to exchange signatures on the seed commitments and instance transcripts. Given the seed commitments, the corresponding openings, and the agreed transcripts, each external party can audit the respective semi-honest instance and check if a party deviated from the intended behavior. Signatures on the commitments and the transcripts make this data publicly verifiable and non-reputable. However, PVC secure protocols have the problem that aborts are not publicly verifiable. An adversary cheating in a particular semi-honest instance that is selected for auditing can abort before opening its seed commitment. This prevents both the local and public auditing of this instance. In covert protocols, this problem is addressed by interpreting an abort in this phase

3. Financially Backed Covert Security

as cheating, a workaround that is not available to PVC secure protocols.

For the design of PVC secure protocols, it is, therefore, necessary to force parties to provide all data necessary to audit the selected protocol instances (and create a PoM) before they learn the selection. In previous work [12, 70, 116, 126, 182], this has been achieved through the use of oblivious transfer (OT). Each party samples a watchlist of s instances and receives the corresponding seed openings from each other party via OT. As each party has its own watchlist, we call these watchlists *local*. The OT hides the watchlist from the sender, thereby forcing the sender to disclose the openings without learning whether a potential cheating attempt will be detected. By exchanging signatures on the OT transcript, the OT becomes publicly verifiable such that invalid openings inserted into the OT become publicly auditable as well.

The OT-based approach has one severe downside. Recall that there needs to be at least one unaudited instance to provide the precomputation for the online phase. When relying on local watchlists, it is therefore necessary to set $t > s \cdot n$ in order to prevent the auditing from covering all instances. This restricts the deterrence factor by $\frac{1}{n}$ from above. In [70], the authors suggest that, in the input-independent setting, this restriction can be circumvented by repeating the PVC secure protocol until one instance is not part of any local watchlist. Another workaround to this problem employed in the two-party case for protocols based on garbled circuits [12, 116, 126, 182] is to utilize an asymmetric semi-honest base protocol that is maliciously secure in regard to cheating attempts of one party and semi-honest secure in regard to cheating attempts of the other party. The malicious security of the protocol in regard to the first party makes it impossible for the first party to execute a successful cheating attempt. Therefore, it suffices for the first party to audit the second one. The resulting protocol has just a single watchlist, the one of the first party. Therefore, it is possible for the watchlist to have size $s = t - 1$, which results in a deterrence factor of $\frac{t-1}{t}$.

3.1. Our Contribution

In this thesis, we aim to transform arbitrary semi-honest secure protocols into covert secure protocols with efficient financial punishment of detected cheaters. In the following, we call such protocols *financially backed covert (FBC)* secure. As financially backed covert security naturally builds on top of publicly verifiable covert security – misbehavior needs to be publicly verifiable to be punishable by an external judge – we envision the transformation to be performed in two steps: one from semi-honest security to PVC security and one from PVC security to FBC

3. Financially Backed Covert Security

security. We make significant contributions in both steps and present an additional efficiency improvement for covert secure protocols that straightforwardly translates to PVC and FBC security.

First, we present a generic compiler that transforms an arbitrary semi-honest secure multiparty protocol into a PVC secure one. While the only prior work on PVC considering the multiparty setting [70] either restricts the deterrence factor to be smaller than $\frac{1}{n}$ or requires protocol repetitions, that significantly increase the protocol complexity, our transformation does not suffer from either of the two restrictions. Moreover, together with concurrent work [155], we are the first to provide a formal specification and security proof for a PVC secure multiparty protocol. Our work has been disseminated in the following article, which can be found in Appendix A.

- [88] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Generic Compiler for Publicly Verifiable Covert Multi-Party Computation”. In: *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*. 2021, pp. 782–811. **Part of this thesis.**

Second, we formally define the notion of *financially backed covert (FBC)* security and present transformations compiling arbitrary PVC secure protocols into FBC secure ones. Our notion requires parties to lock a security deposit at the beginning of the protocol at a judge, who disburses the deposits after the protocol execution to all parties except those that have been proven to be corrupted. The judging process in our protocols is highly optimized so that the judge can be efficiently realized via a smart contract. We showcase practicability by providing a prototype implementation of the judge as an Ethereum smart contract. Our work has been published in the following article, which can be found in Appendix B.

- [86] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Financially Backed Covert Security”. In: *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part II*. 2022, pp. 99–129. **Part of this thesis.**

Third, we identify redundancy in the current approach of applying the cut-and-choose technique to protocols in the offline/online model. Prior work considering the offline/online model combines a covert secure offline phase with a malicious secure online phase. The rationale behind this approach is that the online phase is highly efficient, even when instantiated with malicious security, such that a reduction of the security level would only have a negligible impact. However, this rationale does not cover the whole picture. Although a reduction of the security

3. Financially Backed Covert Security

level of the online phase does not significantly impact the efficiency of the online phase, it can significantly reduce the requirements on the precomputation material generated by the offline phase. In order to address this redundancy, we first prove that the combination of a covert secure offline phase with a covert secure online phase yields a covert secure protocol with a deterrence factor equal to the minimum of the deterrence factors of the two individual protocols. We then demonstrate the impact of our observation using the TinyOT [146] protocol as an example. Our evaluation shows that the downgrade of the online security to covert security reduces the communication complexity of the offline phase by approximately 35%. Our work resulted in the following publication, which can be found in Appendix C.

- [90] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Putting the Online Phase on a Diet: Covert Security from Short MACs”. In: *Topics in Cryptology - CT-RSA 2023 - Cryptographers’ Track at the RSA Conference 2023, San Francisco, CA, USA, April 24-27, 2023, Proceedings*. 2023, pp. 360–386. **Part of this thesis.**

3.2. Key results

In the following, we provide a more detailed discussion of our key results in the publications outlined above. We refer the reader to our published articles [86, 88, 90] (Appendix A-C) for further details.

3.2.1. Generic Compiler for Publicly Verifiable Covert Security

In [88], we present a generic compiler transforming an arbitrary semi-honest secure multiparty protocol into a PVC secure protocol. Instead of relying on oblivious transfer, inherently restricting the deterrence factor by $\frac{1}{n}$, we propose an alternative approach relying on time-lock puzzles [154]. In the following, we will focus on input-independent offline protocols that are combined with a malicious secure online protocol. However, the explained techniques straightforwardly translate to the input-dependent setting.

The PVC compiler. According to the cut-and-choose blueprint introduced above, our compiler requires the parties to first execute t instances of the semi-honest secure offline phase with precommitted randomness and agree on the instance transcripts. Next, the parties run a maliciously secure subprotocol Π_{PC} , that takes the openings of all seed commitments (one per party and instance) as input, samples a random index $r \in [t]$ and defines the output tuple to be r and the seed openings of all instances except the r -th. Instead of releasing the output directly, Π_{PC}

3. Financially Backed Covert Security

releases the output in two phases. In the first phase, Π_{PG} locks the output tuple into a time-lock puzzle and releases the puzzle to all parties. After receiving the puzzle, the parties exchange signatures on the commitments, the instance transcripts, and the puzzle. The hardness of the time-lock puzzle is set such that an adversary cannot solve the puzzle before the signature exchange times out. In case of an abort during or prior to the signature exchange, honest parties terminate the protocol themselves without further investigating for adversarial misbehavior. After a successful signature exchange, the parties continue with the second output phase of Π_{PG} , which releases the output tuple in clear. In case of an abort during the second phase of outputs, the parties solve the time-lock puzzle. Either way, they receive the index r and the seed openings; the second phase of outputs just avoids the need to solve the puzzle in the optimistic case. The parties then use the received information, i.e., the seed commitments, seed openings, and instance transcripts, to audit all semi-honest instances but the r -th. Note that the index r defines one global watchlist for all parties in contrast to the OT-based approach, in which each party samples its own local watchlists. If the auditing does not yield any misbehavior, the parties interpret the output of the unaudited r -th semi-honest instance as the output of the PVC secure offline phase. In case of misbehavior, either due to deviations in one of the semi-honest instances or due to invalid seed openings submitted to Π_{PG} , parties create a PoM and terminate the protocol execution. If the auditing indicates a deviation from the protocol in one of the semi-honest instances, the PoM contains the seed commitments, seed openings and instance transcripts. If the auditing indicates an invalid opening submitted to Π_{PG} , the PoM contains the seed commitments and the time-lock puzzle.² In both cases, the PoM is made publicly verifiable and non-reputable by appending the corresponding signatures of the accused party. As each party audits $t - 1$ randomly selected instances out of the totally executed t instances, the protocol provides a deterrence factor of $\frac{t-1}{t}$.

The rationale for the utilization of time-lock puzzles is that a signature on the puzzle commits a party to the locked secret in a publicly verifiable way while simultaneously guaranteeing that the secret is hidden until after the signature exchange. The latter prevents parties from basing their decision whether to sign or abort on the locked secret. This way, we force parties attempting a cheating attempt to disclose the data that can potentially be used to prove their misbehavior without knowing whether their cheating attempt will be detected. Even though the adversary can abort without sending a signature, it has to decide independently

²The parties can relieve the judge of the necessity to solve the time-lock puzzle by solving the puzzle themselves and attaching the solution and a correctness proof to the PoM. This is made possible by the use of verifiable time-lock puzzles (cf. Section 2.3.)

3. Financially Backed Covert Security

of the detection event. Such an abort is tolerated in PVC security.

Puzzles generation. For the sake of clarity, we have simplified the above description by abstracting over some of the details of the time-lock puzzle generation.

First, time-lock puzzle schemes require a setup generating public parameters. In common instantiations based on [154], the setup yields trapdoor information, which can be exploited to extract the secret locked within a puzzle efficiently. Therefore, we require a trusted setup that prevents any potentially malicious party from learning the trapdoor information. However, the setup needs to be executed only once. The generated public parameters can be used for an unlimited number of protocol instances. Therefore, we propose the setup to be realized with a distributed protocol executed between members of a committee of well-accepted authorities, such as governmental institutions or universities. An alternative approach could be to investigate whether a time-lock puzzle scheme matching our requirements can be instantiated based on hidden order groups with a public setup, such as ideal class groups of imaginary quadratic fields [44] or Jacobians of hyperelliptic curves [77]. The public setup property would ensure that the public parameters are generated efficiently without yielding a trapdoor to the scheme.

Second, we assume Π_{PG} to be instantiated via general-purpose maliciously secure multiparty computation. Common instantiations of the TLP primitive require two exponentiations in an RSA group to generate a time-lock puzzle, which is highly expensive when executed within general-purpose MPC. Therefore, we propose an improvement to Π_{PG} , in which parties precompute what we call *empty* puzzles, i.e., time-locked random masks without any locked secrets. Parties then use these as additional input to Π_{PG} . Instead of generating a new puzzle, Π_{PG} only combines the provided empty puzzles and uses the result to hide the actual secret. This way, we eliminate exponentiations within Π_{PG} . However, our modification enables malicious parties to modify the resulting puzzle by inputting non-empty puzzles. Fortunately, we can interpret such an attack as a simple unpunished abort analogously to an abort during the signature exchange. Recall that the definition of PVC security (cf. Section 2.2.2) allows the adversary to perform an *early* abort, i.e., abort before it learns whether a potential cheating attempt is successful. During the TLP generation, the adversary has to make the decision to insert malformed empty puzzles before learning anything about the watchlist and, hence, about the detection of a potential cheating attempt. Therefore it is tolerable to interpret a malformed empty puzzle as an early abort as long as we can ensure that such an attack is always detected. To enable parties to detect the described attack reliably, we utilize two techniques. First, we encode the secret locked within the puzzle such that any non-empty puzzle inserted in Π_{PG} renders

3. Financially Backed Covert Security

the solution of the resulting combined puzzle invalid. Second, we require parties to provide the randomness used to compute the empty puzzles to Π_{PG} and release this randomness with the second round output of Π_{PG} . This allows parties to recompute the puzzle generation and verify if the received puzzle is correct. The first technique allows detection of the attack if the adversary aborts before the second round output and the parties have to solve the puzzles. The second technique allows detection of the attack if the parties do not have to solve the puzzle, i.e., if the second round output is available.

Concurrent with our work, [155] proposed a transformation from semi-honest security to PVC security that also utilizes a malicious secure subprotocol and time-lock puzzles. However, instead of computing the time-lock puzzle within the subprotocol, they compute a secret sharing of the puzzle payload within the subprotocol and require each party to generate its own time-lock puzzle, with its own public parameters, on its share of the payload. This way, the authors avoid the necessity of a trusted TLP setup and improve the efficiency of the subprotocol. However, on the downside, their approach requires the judge to solve multiple time-lock puzzles in order to verify a PoM.

Security. Together with concurrent work [155], we are the first to provide a formal security proof for a PVC secure protocol in the multiparty setting. We refer the reader to our paper [88] (Appendix A) for the full formal proof. In the following, we focus on the major challenge of our security proof, the reduction to the security guarantees of time-lock puzzles in a simulation-based security proof.

In fact, we are the first to incorporate time-lock puzzles into a simulation-based security proof. An important detail when using time-lock puzzles is that the puzzle does not actually hide any secrets. Any party can start solving any puzzle after the protocol termination and will eventually learn the locked secrets. Instead, the puzzle prevents parties from reacting to the secrets locked within a puzzle. Time-lock puzzles can, therefore, only be used to show that some actions taken by a party up to some given round, e.g., the messages sent by the party, are independent of the secret locked within a puzzle. Based on this insight, we design a novel proof strategy that allows us to prove the indistinguishability of two hybrid worlds via a reduction to the security guarantees of the time-lock puzzle scheme. We outline this strategy in the following.

Consider a scenario in which there are two hybrid worlds H_1 and H_2 with the following structure. Both worlds send a puzzle p_0 locking a secret bit s_0 to the adversary, execute the adversary up to some round R , and observe the adversary's behavior B_0 , i.e., whether the adversary aborts or not. Afterward, both worlds rewind the adversary, send a puzzle p_1 locking a secret bit s_1 to the adversary,

3. Financially Backed Covert Security

and again observe the adversary’s behavior B_1 up to round R . Both puzzles, p_0 and p_1 , are parameterized such that they cannot be solved before round $R + 1$. Both worlds keep rewinding the adversary until $B_1 = B_0$ and, only then, finalize the experiment. Once an upper bound of rewinding iterations is reached, the experiment aborts with an error symbol \perp . The secret bit s_1 is the same in both hybrids. Further, in H_2 it holds that $s_0 = s_1$. However, in H_1 it holds that $s_0 \neq s_1$. This is the only difference between the two hybrids.

On an intuition level, it is clearly evident that an adversary capable of breaking the security of the time-lock puzzle is capable of forcing a discrepancy in the output distribution of the two worlds. If the adversary is capable of breaking the time-lock puzzle, it can decide on behavior B_i based on the secret bit s_i (for $i \in \{0, 1\}$), e.g., abort only if $s_i = 0$. As $s_0 \neq s_1$ in H_1 , the adversary can reliably force experiment failures (output \perp) in H_1 . This is not possible for the adversary in H_2 , as the adversary’s view in the main thread and the rewind thread is the same ($s_0 = s_1$). However, given that the adversary is not able to solve the time-lock puzzle before round R , it is not possible for the adversary to select B_i dependent on s_i except with negligible probability. Therefore, it cannot force experiment failures in either of the two worlds. More generally, as B_i is selected independently of s_i except with negligible probability and the rewind thread is exactly the same in both worlds, the worlds need to be computationally indistinguishable.

More formally, our proof strategy works in two steps. First, we make the claim that the action B_0 taken by the adversary is independent of the hybrid world (H_1 or H_2). This claim can be reduced to the security guarantees of the time-lock puzzle scheme as it considers the time-restricted event B_0 . Second, we use the prior claim to show the indistinguishability of the two worlds. As B_0 is the only value that carries over from the initial execution to the rewinding, the distribution of B_0 is computationally indistinguishable in both worlds, as shown by the previous claim, and the steps in the rewinding threads are the same in both worlds, it needs to follow that the outputs of the two worlds are computationally indistinguishable.

3.2.2. Financially Backed Covert Security

In [86], we introduce the notion of FBC security and present three compilers transforming different classes of PVC secure protocols based on the cut-and-choose technique into FBC secure protocols. Class 1 comprises input-independent PVC secure protocols in which parties receive a public transcript of message hashes. Class 2 comprises input-dependent PVC secure protocols in which parties receive a public transcript of messages. Class 3 comprises input-independent PVC secure protocols without a public transcript. As we envision the financial penalties to

3. Financially Backed Covert Security

be realized via a smart contract, we design our protocols to provide a highly efficient punishment procedure, i.e., to minimize the computation required by the smart contract to verify malicious behavior. Since cheating detection is inherited from the transformed PVC secure protocol, we will focus on the punishment in the following.

Definition of FBC security. A financially backed covert secure protocol is comprised of three components: a base protocol π , an evidence algorithm **Blame**, and a punishment protocol **Punish**. It considers three entities: a ledger capturing the notion of money, a judge responsible for the financial punishment, and the set of parties that aims to compute a function with FBC security.

The ledger stores the balances of all parties in the system and allows parties to transfer money between each other. While the definition of the ledger is aligned with the blockchain setting, it does not need to be instantiated via a blockchain-based cryptocurrency. As we do not require the balances of the parties to be public, the functionality of the ledger can also be captured by other monetary systems, e.g., the amount of cash in circulation. The judge is responsible for locking security deposits of all parties throughout the protocol execution and reimbursing those not proven malicious after the protocol execution. The judge does not participate in the protocol execution itself.

Protocol π is a multiparty protocol computing function f . We require a specific structure from π . At the beginning of π , all parties send a fixed amount of coins to the judge. If a party fails to do so, the other parties abort the protocol and the judge redistributes the deposits. At the end of π , each party executes **Blame** with its entire view as input and broadcasts the certificate generated by the algorithm. The certificate contains all information required to prove the misbehavior of a malicious party, if cheating is detected, and to protect against false accusations of malicious parties. After completion of π , the parties and the judge engage in protocol **Punish**. The parties use the previously generated certificates as input. The judge does not have any input. As a result of the **Punish** protocol, the judge refunds all parties that have not been proven malicious.

Security-wise we require three properties: *Simulatability with ϵ -deterrent*, *financial accountability* and *financial defamation freeness*. Simulatability requires that protocol π is a covertly secure protocol with a deterrence factor of ϵ . As π includes the broadcast of the certificates that are the only input to the **Punish** protocol, covert security of π implies that the subsequent punishment protocol does not leak any private data. Financial accountability states that if there is one honest party that detects a cheating attempt in π , then there has to be at least one corrupted party that does not receive its deposit back. Financial defamation freeness

3. Financially Backed Covert Security

states that an honest party always receives its deposit back.

Modeling the PVC secure protocol. We model the protocol specification of the semi-honest protocol utilized by the underlying PVC secure protocol as a tuple of deterministic round algorithms $\{\text{ComputeRound}_*^i\}$, one for each party and round. Round algorithm ComputeRound_k^i takes as input the messages party P_i received in round $k - 1$ and the intermediate state of party P_i computed in the previous round and returns the messages party P_i is supposed to send in round i and a new intermediate state. For simplicity, we assume that each party sends a message to all other parties. As the round algorithm is deterministic, we require all random choices to be derived by a seed embedded into the previous state. The initial state of each party consists of the initial random seed and, if existent, the party's input to the protocol.

Our transformations rely on some features to be provided by the underlying PVC secure protocol. First, we require the PVC secure protocols in all three classes to be based on the cut-and-choose technique, as explained above. For Class 1 and Class 3 protocols, which are input-independent, parties execute t instances of a semi-honest protocol and audit all parties in s of the instances. For Class 2 protocols, which are input-dependent, parties execute one instance of a semi-honest $(n \cdot t)$ -parties protocol, in which each real party simulates t virtual parties and audits s virtual parties of each real party. Second, we require that the PVC secure protocol, if not aborted prematurely, provide means to efficiently derive the initial state of the audited parties in the audited instances in a publicly verifiable way, e.g., via a signed commitment and the corresponding opening, or to prove misbehavior in the provision of the initial states, e.g., via a time-lock signed puzzle containing an invalid opening and a proof allowing the efficient opening of the puzzle. Third, we require the availability of a public transcript of message (hashes) for some of the classes. If a party P_h sends a message $m_k^{(h,i)}$ to another party P_i in round k of a semi-honest protocol instance, it also appends the (hashes of) all messages it is supposed to send to the other parties in round k , i.e., $\{m_k^{(h,j)}\}_{j \notin \{h,i\}}$. After the execution of the instance, the parties aggregate all messages (hashes) into a Merkle Tree and exchange signatures on the tree root. This way, they agree on the transcript and obtain evidence that can be verified by an external judge. If they disagree, they terminate the protocol with a simple abort and each party receives its deposit back. Finally, we ensure that all signatures used throughout the protocol are uniquely bound to a specific purpose, i.e., a signature on the transcript of the i -th semi-honest instance cannot be passed off as the signature on the transcript of the j -th instance. This can be achieved by prepending data with unique identifiers before being signed. We remark that the PVC secure protocols

3. Financially Backed Covert Security

generated by our compiler presented in [88] satisfy all of the required features.

Intermediate state commitments. In all transformations, we extend the underlying PVC secure protocol by requiring parties to commit to their intermediate states in the semi-honest protocol instances. More precisely, we extend the round algorithm ComputeRound_*^m such that it also computes a commitment on the new intermediate state and appends the commitment to all computed messages. The randomness is still derived from the random seed embedded into the previous state. After the execution of the semi-honest protocol instance, the parties compute a Merkle Tree over all received state commitments and exchange signatures on the tree root. This way, they agree on the internal states and obtain evidence that can be verified by an external judge. If they do not agree, they terminate the protocol with a simple abort and each party receives its deposit back.

Punishment in Class 1 and Class 2 protocols. Malicious behavior in a semi-honest instance implies that there is a (virtual) party P_m and a round k such that one of the messages sent by P_m in round k does not match the result of ComputeRound_k^m as emulated during auditing. When encountering misbehavior during auditing, the honest parties accuse the first such protocol deviation via the judge. If there has not been any accusation before a specific deadline, the judge returns the deposits of all parties.

To verify malicious behavior and, hence, perform the punishment, the judge needs to receive P_m 's incoming messages \mathcal{M} of round $k-1$, P_m 's intermediate state of the previous round state_{k-1} , and the hash of the malicious outgoing message m^* , together with evidence proving authenticity of this data. Based on the received data, the judge recomputes the round algorithm ComputeRound_k^m and compares the result with m^* . If the recomputed message does not match m^* , the judge punishes P_m by retaining its security deposit. All other parties are refunded.

Honest parties provide the data required to verify misbehavior and prove its authenticity as follows: In Class 1 protocols, the incoming messages and intermediate state commitments are recomputed by emulating the complete semi-honest instance. In Class 2 protocols, the honest parties only emulate the audited parties and assume all messages from unaudited parties to be correct. As the parties accuse the first protocol deviation, all messages, intermediate states, state commitments, and openings exchanged and computed earlier to round k are correct and, hence, can be recomputed. The authenticity of \mathcal{M} as well as m^* is proven via the signed Merkle root of the transcript tree and Merkle proofs for each message. Authenticity of state_{k-1} is proven via the signed Merkle tree root of the state

3. Financially Backed Covert Security

commitment tree, the commitment of state_{k-1} , its Merkle proof, and the emulated opening.

If $k = 1$, there is no commitment of state_{k-1} as state_{k-1} is the initial state. In this case, the parties prove the authenticity of state_{k-1} to the judge via the means provided by the PVC secure protocol, e.g., a signed commitment and the corresponding opening. Analogously, the parties utilize the means of the PVC secure protocol to prove adversarial behavior in the provision of the initial state, e.g., a signed time-lock puzzle containing an invalid opening and a proof allowing the efficient opening of the puzzle.

The bisection search. The major technique for the transformation of Class 3 protocols is the bisection search, introduced by [51] and popularized in the blockchain space by [78, 117, 120]. The bisection search allows a judge to efficiently identify the first deviation in two lists held by two parties. To do so, the parties compute a Merkle tree over their lists, in which they identify the left-most deviation layer by layer. Figure 3.1 illustrates an exemplary bisection search between two parties, Alice and Bob, holding two lists of size four that deviate in the last element.

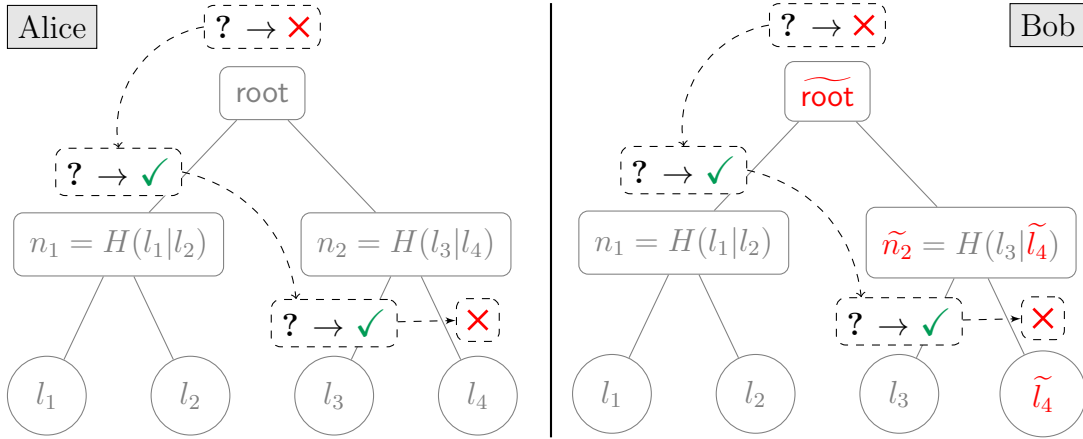


Figure 3.1.: Exemplary bisection search over two lists deviating in the last element.

The parties first submit the roots of their respective trees to the judge. A disagreement in the roots indicates a deviation in the two lists. Next, the parties identify the subtree (left or right) in which the first deviation occurs. To do so, both parties submit the left child of the root (in the example n_1) together with the corresponding Merkle proof. Disagreement on the left child indicates that the first deviation is in the left subtree. Agreement indicates that the first deviation is in the right subtree. This step is repeated recursively for the identified subtree until

3. Financially Backed Covert Security

the search reaches a leaf node. In the example, the parties agree on the left node n_1 and, therefore, try to identify the first deviation in the right subtree below n_2 by comparing the left child of n_1 , i.e., l_3 .

The result of the search is the last element l_{k-1} (in the example l_3) both parties agree on and the index k (in the example 4) of the first deviation in the two lists. The number of rounds it takes to identify the disputed element is logarithmic in the list size.

Transformation of Class 3 protocols. Due to the lack of a public transcript, there are no Class 3 protocols that are PVC secure. However, we can remove the public transcript from PVC secure protocols of Class 1 to get a preliminarily insecure Class 3 protocol and then transform the resulting protocol into one that is FBC secure. The latter is made possible by the interactivity of the punishment protocol, which is not available in PVC security. Note that communication-wise Class 3 protocols are more efficient than Class 1 protocols as Class 3 protocols avoid the necessity of sending each message hash to each other party.

In Class 3 protocols, each party signs each of its outgoing messages. During auditing, each party emulates all the audited semi-honest instances and compares the computed messages to the received ones. In case of a deviation, a party identifies the first such deviation and submits the identifiers (instance index, round index, party index) to the judge. If there is no accusation before a specific deadline, the judge returns the deposits of all parties. Otherwise, the judge selects the deviation with the lowest indices and initiates a dispute between the *blamer*, the party that submitted the deviation, and the *defender*, the party that has been accused. Note that submitted deviations are not necessarily caused by malicious behavior but could also be caused by a previously received maliciously generated message to which the auditing party does not have access. Therefore, it is necessary to select the first deviation to be disputed.

The dispute starts by requiring both the blamer and the defender to create a Merkle tree over the message history up to the disputed round and submit the tree's root to the judge. If both parties agree on the message history, the blamer has all the data to prove misbehavior. The incoming messages of the defender in the disputed round can be submitted and verified with respect to the agreed Merkle root. The maliciously crafted outgoing message is verified via the defender's signature. The intermediate state is submitted and verified analogously to Class 1 and Class 2 protocols.

If the parties disagree on the message history, they run a bisection search, determining the first deviation in their histories. Let m^* be the blamer's claim for the first disputed message, P_i its sender, and k its round. In order to determine

3. Financially Backed Covert Security

the party that submitted the wrong message history, the judge recomputes the disputed message and punishes the party that lied about the message history. The blamer provides the necessary data for the recomputation, i.e., the incoming messages and the intermediate state of P_i in round $k-1$. Authenticity of the incoming messages is proven via Merkle proofs computed with respect to the blamer’s message history. As both parties agree on all messages exchanged before round k , the judge can assume all messages in the blamer’s history before round k to be correct. The authenticity of the intermediate state is proven and verified analogously to Class 1 and Class 2 protocols.

If any party does not provide the requested evidence throughout the punishment protocol, it is punished itself. This is required to prevent a malicious party from utilizing an early false accusation to avoid its own punishment. If the punishment requires the judge to verify initial states or misbehavior while providing the initial states of the audited instances, the data is verified by the judge analogously to Class 1 and Class 2 protocols.

Efficiency improvements and evaluation. In addition to the transformations introduced above, we propose two practical improvements applicable to all three transformations. First, the bisection technique cannot only be applied to dispute a message history but also to dispute computations. Instead of requiring the contract to recompute complete algorithms, e.g., a full round algorithm, the computation can be outsourced to the disputing parties (the blamer and the defender). To identify the disagreement on the computation, the two parties run a bisection search, yielding a single arithmetic gate on which the parties disagree. The judge then only verifies this single gate. Second, it is not necessary for the judge to be aware of all computations potentially required to verify misbehavior, e.g., all rounds algorithms. Instead, we can provide the required code in an ad-hoc way. To do so, all parties sign the root of a Merkle tree describing all computations. Once the judge needs to verify a specific protocol step, the parties submit the step information, e.g., an arithmetic gate or a round algorithm, and proof authenticity via a Merkle proof.

In order to show the feasibility of a smart contract-based punishment, we have implemented the judge for Class 3 FBC secure protocols as an Ethereum smart contract. Class 3 FBC secure protocols involve the most complex punishment protocol of all of our transformations. Our implementation includes the engineering improvements discussed above. We implement the judge in a way that allows us to utilize a single contract for arbitrarily many FBC secure protocols. Our evaluation includes the optimistic scenario, in which there are no detected cheating attempts, and pessimistic scenarios, in which there is detected cheating, with

3. Financially Backed Covert Security

different parameters, i.e., number of parties, message sizes, protocol rounds, and complexity of the round algorithm. In the optimistic case, parties need to register the protocol instance, deposit the security deposit, and withdraw the deposit after protocol execution. For three parties, this accumulates to a total (over all parties) of 533 K gas, which costs approximately \$17 USD at the time of writing [84]. In the worst case, a protocol execution with three parties, ten semi-honest instances, ten communication rounds, 320-byte messages, and round algorithms with up to 1 000 arithmetic gates accumulates a gas consumption of 2 412 K gas, which costs around \$76 USD at the time of writing [84].

3.2.3. Covert Security From Short MACs

Previously proposed covert secure protocols in the offline/online model are composed of a covert secure offline phase and a malicious secure online phase. Due to the high efficiency of the online phase, it is assumed that a reduction of the security level of the online phase does not impact the overall protocol efficiency significantly. In [88], we challenge this belief. While we agree that the reduction of the online phase’s security does not significantly impact the online phase’s efficiency, we observe that it can significantly reduce the requirements on the precomputation material generated by the offline phase.

One particular upside of our observation is that, security-wise, the efficiency gain is for free. Intuitively, this is because an adversary only needs to cheat once to break security. Therefore, it does not affect security if the adversary can choose the phase it tries to cheat as long as the deterrence factor of both phases’ is the same. In [90], we formally prove this intuition by showing that a protocol composed of a covert secure offline phase with a deterrence factor of ϵ_1 and a covert secure online phase with a deterrence factor of ϵ_2 is itself a covert secure protocol with deterrence factor $\epsilon = \text{Min}(\epsilon_1, \epsilon_2)$.

We showcase the impact of our observation by the example of the famous TinyOT [147] protocol. In the following, we will focus on this case study.

The TinyOT online phase. The TinyOT protocol is a general-purpose two-party protocol for computations over boolean circuits that is based on the secret-sharing approach. In a nutshell, the protocol works as follows: For each input wire of the boolean circuit, the parties receive an XOR-sharing of the wire value, e.g, if wire w_i has value x_i , then we provide a value x_i^A to party A and a value x_i^B to party B such that $x_i = x_i^A \oplus x_i^B$. Subsequently, the parties compute the intermediate wires of the circuit layer by layer. If a wire w_l is the result of an addition gate with input wires w_j and w_k , parties compute a sharing of w_l ’s value x_l by adding their shares of the

3. Financially Backed Covert Security

input wires, i.e., $x_l^A = x_j^A \oplus x_k^A$ and $x_l^B = x_j^B \oplus x_k^B$. If the wire w_l is the result of a multiplication gate, parties compute the secret sharing of x_l interactively using the Beaver [23] technique. Given a fresh multiplication triple provided by the offline phase consisting of a random tuple $(\alpha_A, \beta_A, \gamma_A)$ known to party A and a random tuple $(\alpha_B, \beta_B, \gamma_B)$ known to party B such that $(\alpha_A \oplus \alpha_B) \odot (\beta_A \oplus \beta_B) = (\gamma_A \oplus \gamma_B)$, the parties compute the sharing of x_l as follows: First, each party P ($P \in \{A, B\}$) sends $e_P = (\alpha_P \oplus x_j^P)$ and $d_P = (\alpha_P \oplus x_k^P)$ to the other party. Party P then computes $e = e_A \oplus e_B$, $d = d_A \oplus d_B$ and $x_l^P := \gamma^P \oplus e \odot x_k^P \oplus d \odot x_j^P \oplus e \odot d$. Once the circuit evaluation reaches the output wires, the parties start the output phase, in which they exchange their shares on the output wires to reconstruct the final output.

In order to guard the protocol summarized above against malicious deviations, the TinyOT protocol utilizes information-theoretic message authenticate codes (MAC). Each party P holds a global key $\Delta_P \in \{0, 1\}^\kappa$ where κ is the statistical security parameter. The goal is that for each wire share x_i^P , party P holds a MAC $M_i^P \in \{0, 1\}^\kappa$ and the other party \bar{P} holds a local key $K_i^{\bar{P}} \in \{0, 1\}^\kappa$ such that $M_i^P = K_i^{\bar{P}} \oplus x_i^P \odot \Delta_{\bar{P}}$.

To authenticate an input bit x of party A , the parties make use of an authenticated random bit produced by the offline phase, i.e., the offline phase provides $(r, M[r])$ to A and $(K[r])$ to B . To transform the authenticated bit into an authenticated secret sharing of A 's input bit x , A sends $c = x \oplus r$. Then A defines $(x_A, M[x_A], K[x_B]) = (r, M[r], c \odot \Delta_A)$ and B defines $(x_B, M[x_B], K[x_A]) = (c, 0^\kappa, K[r])$. Due to the additive homomorphism of the MACs, it is possible to compute the local keys and MACs associated with intermediate wires symmetrically to the shares of the wire values. However, the multiplication triples received by the offline phase need to be authenticated via MACs as well.

Whenever a party reveals a share, during the multiplication procedure or the output phase, the share is authenticated to the receiver via the sender's MAC on the share. Given the received share, the receiver recomputes the MAC and compares it to the received one. In practical instantiations, parties do not actually send the MAC with every share they reveal but authenticate all shares exchanged during the protocol via an aggregated MAC check. Throughout the protocol, each party maintains a list of MACs it should have sent and a list of MACs it should have received. Before the parties open the output wires, they send the hash of their outgoing MAC list to the other party, which verifies the hash against its own list. The same aggregation technique is used for the shares exchanged in the output phase.

Note that a part can only cheat in this protocol by flipping the bits of shares exchanged during the protocol execution. Such a deviation can only go undetected

3. Financially Backed Covert Security

if the adversary manages to guess the MAC for the modified bit correctly, which is equivalent to successfully guessing the other party's global key. As the guess cannot be verified locally, the adversary has only one shot for this guess. It follows that the probability of a successful attack on the protocol is $\frac{1}{2^\kappa}$.

Covert security from short MACs. We observe that the TinyOT online phase with a decreased statistical security parameter t is almost a covert secure protocol with deterrence factor $\epsilon = \frac{1}{2^t}$. This is because the only attack on the TinyOT protocol requires the attacker to guess the other party's randomly sampled global MAC key, which has t bits.

However, there is a problem when proving covert security of such a modified protocol. In the final output round, both parties exchange the shares of the output wires and append the corresponding MACs (or the hash of the MAC list). As we assume a rushing adversary, we allow the adversary to receive the honest party's message first and then send its message. The adversary, therefore, learns its output before deciding whether to cheat by sending modified output shares. While the prevalent notion of covert security, the one we have considered so far, which is called the strong explicit cheat formulation (SECF), allows an adversary to abort after learning its output, it does not allow the adversary to attempt cheating after learning its output. We propose two approaches to tackle this problem.

First, we propose an alternative notion of covert security called *intermediate explicit cheat formulation (IECF)*. This notion still requires the adversary to make its cheating decision independent of its output, as in the SECF, but allows the adversary to learn its own output even in the case that cheating is detected. We deem the security downgrade justifiable as the adversary only receives information that it would also receive by behaving honestly. To transform the TinyOT protocol with short MACs into an IECF covert secure protocol, we split the output phase into two rounds. In the first round, parties exchange commitments on the output shares and MACs. In the second round, they open the commitments. The commitment prevents the adversary from lying about its output shares after learning its output.

Second, we propose an extension to the TinyOT protocol that allows us to prove the security according to the SECF. We adapt the circuit computing function f so that the output wires hold a commitment to the actual outputs and the corresponding opening. The output phase is again split into two rounds. In the first round, the parties reconstruct the commitment and verify its authenticity as usual. In the second part, the parties reconstruct the opening and use it to open the commitment. An adversary only learns the output in the second round. However, due to the commitment exchanged earlier, it is impossible for the adversary to send

3. Financially Backed Covert Security

incorrect opening shares without being detected. This allows the honest party to interpret incorrect shares received in the second round as an abort. The downside of this second approach is that the computation of the commitments as part of the jointly evaluated circuit imposes a significant efficiency overhead to the online phase. Therefore, we assess the first approach to be more practical.

Evaluation. Due to the aggregated MAC check in the TinyOT protocol, the reduction of the MAC length does not positively impact the communication complexity of the online phase. In fact, the covert online phase, as presented above, is slightly less efficient than the malicious secure one due to the commitments exchanged during the output phase. We deem the reduced computation and storage complexity as insignificant. However, we predict that the offline phase required for the covert secure online phase is significantly more efficient than the offline phase required for the malicious secure online phase.

In the TinyOT protocol, the offline phase is responsible for providing authenticated bits for the input phase and secret shared authenticated multiplication triples for the efficient evaluation of multiplication gates. In the boolean setting, these are typically provided by protocols based on oblivious transfer, as, for example, done in [122, 146, 147, 171, 178]. Due to the lack of an explicit specification of a covert secure offline phase providing the required precomputation material, we design the offline phase ourselves. The offline protocol employs the cut-and-choose approach introduced above in order to transform a semi-honest secure protocol into a covert secure one. The semi-honest instances utilize oblivious transfer to generate authenticated bits and employ techniques from [122, 171] to transform authenticated bits into authenticated triples. We instantiate the OT based on [118]. In our evaluation, we focus on the communication complexity as communication constitutes the major bottleneck of the offline protocol. We compute the communication complexity for both the long MAC and the short MAC version of the precomputation with different deterrence factors and different numbers of generated triples. Our results yield that for deterrence factors $\epsilon \leq \frac{7}{8}$ the communication complexity of a protocol generating precomputation material for a covert secure online phase is at least 35 % lower than the complexity of a protocol generating precomputation material for a malicious secure online phase. The influence of the number of generated triples is insignificant.

3.3. Related Work

Research on general-purpose multiparty computation has been initiated by [24, 26, 55, 104, 180]. Since then, there have been tremendous efforts to improve MPC in many different settings, especially in the malicious security setting. Today, the most relevant directions are authenticated garbling, e.g., [76, 122, 171, 178], the TinyOT family, e.g., [46, 96, 131, 146], and the SPDZ family, e.g., [65, 69, 71, 123].

The concept of covert security was first introduced by Aumann and Lindell [14], who focused on the two-party case. Their approach has been extended to the multiparty setting by Goyal et al. [108]. While the protocols provided by [14, 108] rely on garbled circuits, subsequent work adapted the notion of covert security to different settings. In [67], the authors present a generic transformation from semi-honest security to covert security in the honest majority setting. In [133], the authors extend MPC techniques introduced by the IPS compiler [119] to construct a generic compiler from semi-honest security to covert security in the dishonest majority setting. Their construction inherits excellent asymptotic efficiency from the underlying IPS compiler. In [69], the authors adapt the notion of covert security to the SPDZ protocol [68], which is tailored to evaluating arithmetic circuits in the dishonest majority setting. In [132], the author proposes improvements to cut-and-choose and garbled circuit-based malicious and covert secure two-party computation. Another line of work focuses on adding stronger security guarantees to covert security. While [127, 143] both propose protocols, in which even a successful adversary can only learn up to one bit about the private input of the honest party, the protocol of [143] additionally guarantees correctness even in the case of successful cheating. In [130], the authors add fairness to covert secure two-party computation based on an external third party. The fairness property ensures that if one party learns the output of the protocol execution, the other party will do so as well. Experimental evaluations of covert secure general-purpose MPC protocols have been provided by [68, 151]. Furthermore, there is a line of work adapting the notion of covert security to special purpose protocols such as private set interaction and pattern matching [111], oblivious polynomial evaluation [110] and oblivious transfer [11]. Despite the large amount of research on covert security, all protocols in the offline/online model utilize a malicious secure online phase. We are the first to observe that the overall protocol efficiency can benefit from a reduction of the security level of the online phase.

The notion of public verifiable covert security has been introduced by Asharov and Orlandi [12]. The core technique of [12] is to utilize a novel primitive called *signed oblivious transfer* that ensures that the receiver of the oblivious transfer also receives a signature by the sender on the selected message. In [126], the au-

3. Financially Backed Covert Security

thors present an improved covert secure protocol that is based on a more efficient instantiation of the signed oblivious transfer that is, unlike the signed oblivious transfer presented by [12], compatible with the OT extension technique [118]. In [116], the authors simplify the previous two approaches by proposing a PVC secure protocol that does not rely on signed oblivious transfer. The resulting protocol is significantly more efficient than the previous two protocols. The protocols in [12, 116, 126] all focus on the two-party setting, utilize the cut-and-choose technique, and are based on garbled circuits enabling them to support a deterrence factor of $\epsilon = \frac{t-1}{t}$, where t is the number of semi-honest instances executed in the PVC secure protocol. In [70], the authors extend the scope of PVC security beyond garbled circuits by proposing a generic transformation from semi-honest security to PVC security. While the authors focus on the two-party case, they also provide an intuition on how their ideas can be translated to the multiparty setting. As discussed above, the deterrence factor in their protocol is bound from above by $\frac{1}{n}$. Higher deterrence factors can be supported but require the parties to rerun the protocol until there is one semi-honest instance that has not been audited. Our work [88], in contrast, provides a formal specification of a transformation from semi-honest security to PVC security and does not require protocol repetitions to support a deterrence factor of $\epsilon = \frac{t-1}{t}$. Concurrent to our work on PVC security [88], [155] have proposed a transformation from semi-honest security to PVC security that is based on the same techniques as our compiler, i.e., time-lock puzzles. However, as discussed above, they provide a more efficient way to instantiate the generation of the time-lock puzzles. Further, there has been work adapting PVC security to the use case of private function evaluation [137], presenting a transformation from semi-honest security to PVC security in the honest majority setting [13], and strengthening the security guarantees by ensuring that even a successfully cheating adversary cannot break correctness and can only learn one bit of leakage from the honest party's input [136], akin to what have been proposed by [143] for covert security.

While our work [86] is the first to formally introduce the notion of financially backed covert security, there has been prior [103, 182] also linking detected cheating in PVC secure protocols to financial punishment. In [182], the authors describe a two-party garbled circuit-based protocol in which a smart contract financially punishes misbehavior. The authors construct the protocol in a way that prevents malicious cheating attempts at any step but one, the creation of the garbled circuits. Therefore, it is sufficient to audit this particular step via the cut-and-choose technique. If the evaluator detects a maliciously crafted circuit during auditing, it triggers punishment via a smart contract. To instantiate the judge efficiently, they utilize the bisection technique as well. While our work [86] also aims to instantiate

3. *Financially Backed Covert Security*

the judge as a smart contract and increases the efficiency of the judging and punishment with the bisection technique, we generalize the work of [182] in several directions and overcome significant challenges on the way. While [182] restricts cheating to one message and the algorithm generating that message, we consider misbehavior in interactive protocols. Furthermore, the protocols considered in our work can involve more than two parties, be input-dependent (Case 2), and tolerate private party-to-party communication (Case 3). In [103], the authors proposed to augment PVC security with real-world legal contracts. They prove the security of their scheme concerning rational adversaries, adversaries that will only cheat if the expected revenue of cheating is positive. By showing that the expected cost of detected cheating attempts, enforced via the real-world legal contract, outweighs the expected revenue of successful cheating, they prove that any rational adversary refrains from cheating attempts. However, they restrict their work to a specific protocol, i.e., garbled circuit-based two-party computation, while we consider a more general setting.

4. Statement Oblivious Witness Encryption

Given an NP language \mathcal{L} with associated relation \mathcal{R} , witness encryption, introduced by Garg et al. [99], enables a party to encrypt a message m under a statement x . The ciphertext can only be decrypted by a party holding a valid witness w such that $\mathcal{R}(x, w) = \text{true}$. The applications for witness encryption range from public key encryption with fast key generation [99] and attribute-based encryption for general circuits [99] to using it for encrypting a prize for solving an NP-hard puzzle like the millennium problems [99] or achieving fairness in MPC [57].

Despite the enormous potential of the primitive, it still suffers from severe limitations hindering its deployment in real-world use cases. First, known instantiations supporting general languages rely on strong cryptographic assumptions such as multilinear maps [99, 102, 105, 135], indistinguishability obfuscation [98], or cryptographic invariant maps [34] and lack practical efficiency. Second, the standard notion of witness encryption does not provide any means to keep the statement used for encryption hidden and, hence, rules out further interesting use cases.

In order to address the first shortcoming, Goyal et al. [106] propose an alternative approach, shifting the trust assumption towards assuming that a subset of servers in a committee is honest. The authors present a protocol in which a committee of servers is responsible for storing the plaintexts labeled with a statement in a secret shared way and releasing the plaintexts to clients that can provide a valid witness. This approach is based on a line of work presenting constructions of how such committees can be obtained in a blockchain setting [27, 49, 101]. While the protocol in [106] is solely based on standard cryptographic assumptions, it does not consider the need to keep the statement used for encryption private. In fact, we are the first to observe the potential of private statements. Keeping the statement used for encryption hidden not only enhances existing use cases with additional privacy guarantees but also enables new use cases, e.g., in decentralized finance.

4.1. Our Contribution

In this thesis, we introduce and formally define the notion of *statement oblivious threshold encryption (SO-TWE)*. This notion provides the same functionality as witness encryption but relies on a committee with a fixed threshold of honest servers instead of strong cryptographic assumptions. Moreover, it augments witness encryption with a statement obliviousness property that ensures that the statement used for encrypting a ciphertext remains private, even from the servers performing the decryption. We show two ways SO-TWE can be instantiated and prove the security of our constructions. Our work has been disseminated in the following article and can be found in Appendix A.

- [91] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Statement-Oblivious Threshold Witness Encryption”. In: *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*. 2023, pp. 17–32. **Part of this thesis.**

4.2. Key Results

In the following, we provide an overview of our key results. Most importantly, we formally define the notion of SO-TWE and show two ways to instantiate SO-TWE via multiple transformations. While doing so, we also introduce the notions of *oblivious threshold tag-based encryption* and *anonymous threshold identity-based encryption* together with formal definitions. Figure 4.1 gives an overview of our constructions. We refer the reader to our published article [91] (Appendix D) for further details.

Defining SO-TWE. In SO-TWE, defined for some NP-language \mathcal{L} with relation \mathcal{R} , we envision a setting where a client, Alice, locally encrypts a message m under a statement x with a committee public key pk . The result is a ciphertext c . Another client, Bob, receiving ciphertext c , can submit c to the committee with a statement-witness pair (x', w') . The committee members compute decryption shares d_i (which can also be equal to a special error symbol) with their secret key shares sk_i and return the decryption shares to Bob. Each server does this independently, i.e., there is no communication between the committee members. Once Bob has received s valid decryption shares from different servers, he runs a reconstruction algorithm on the shares, which yields the decryption m' . By correctness, it needs to hold that $m' = m$ if $x = x'$ and $\mathcal{R}(x', w') = \text{true}$.

4. Statement Oblivious Witness Encryption

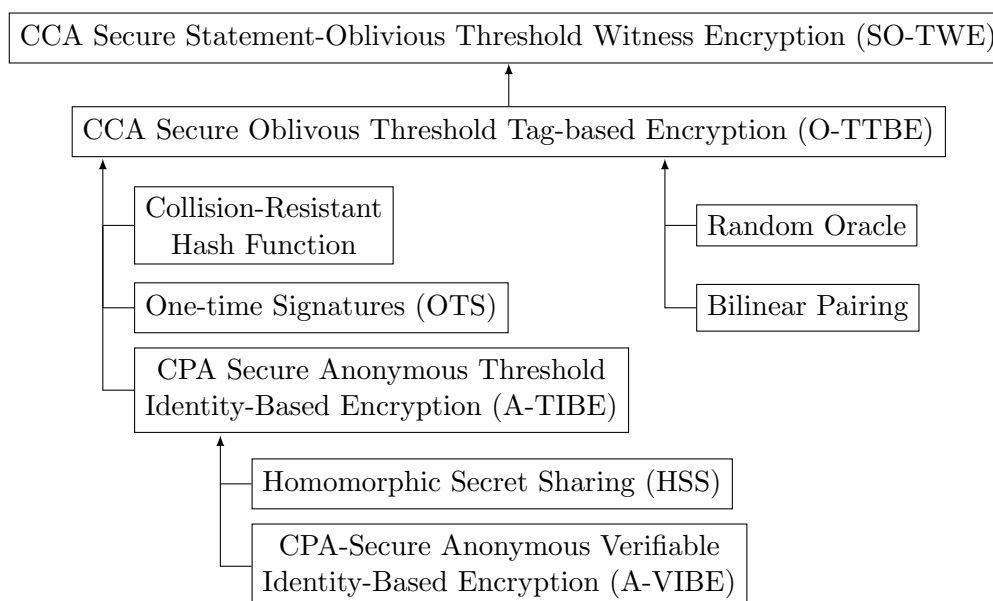


Figure 4.1.: Different ways to instantiate statement-oblivious threshold witness encryption.

Security-wise, we require a SO-TWE scheme to satisfy what we call *indistinguishability and statement obliviousness under chosen-ciphertext attacks*. Intuitively, this property states that any computationally bounded adversary that receives a ciphertext c^* that is the result of encrypting one of two messages, m_0 or m_1 , with one of two statements, x_0 or x_1 , all of which are selected by the adversary, can neither differentiate which message was encrypted nor which statement was used. To account for the threshold setting, we enable the adversary to corrupt up to $s - 1$ of the committee members, i.e., learn their secret key shares. As we envision a client-server setting in which an external committee performs decryption upon request, we require security under chosen ciphertext attacks (CCA). This means the adversary can request decryption shares of arbitrary ciphertexts under arbitrary statements and witnesses, just not for c^* with x_0 or x_1 .

It is important to note that our notion implies that any subset of servers of size smaller than s must not be able to check if a ciphertext c has been created with a particular statement x . This fact leads to the major challenge we face when instantiating SO-TWE. In CCA secure encryption schemes, it is standard to validate received ciphertexts before decryption in order to ensure that the secret key operation is applied to the ciphertext only if the decryption will indeed yield a correct plaintext. This technique is required to prevent an adversary from exploiting homomorphisms in the ciphertext space, i.e., adapt the challenge ciphertext c^*

4. Statement Oblivious Witness Encryption

such that the decryption of the adapted ciphertext yields information about the decryption of c^* . In SO-TWE, we can apply such a validation only to a limited extent. As we require that a single server cannot validate if a ciphertext c has been created with a particular statement x and we do not allow any communication during decryption, servers cannot decline the secret key operation in case of invalid statements. A server has to execute the decryption for whatever arbitrary statement it receives (as long as the witness matches). From the resulting decryption share, we require that it does not indicate if the reconstruction will yield the correct plaintext. Otherwise, this information could be used to test for the correct statement. However, we also require that a decryption share created based on an invalid statement does not yield any information about the correct plaintext.

Instantiating SO-TWE. As a first step towards instantiating SO-TWE, we extend the notion of (threshold) tag-based encryption [9, 139] to *oblivious threshold tag-based encryption (O-TTBE)*. O-TTBE is defined equivalently to SO-TWE but without the necessity of providing or checking witnesses upon decryption; the statement x is now just called tag t . Given an O-TTBE scheme, it is possible to instantiate SO-TWE straightforwardly. The SO-TWE ciphertext is computed using the O-TTBE encryption with the witness as the tag. During decryption, the servers check the received statement-witness pair and perform the O-TTBE decryption only if the witness relation is verified successfully. While this transformation provides a simple instantiation of SO-TWE, it shifts the challenge outlined above to the instantiation of the O-TTBE scheme.

We provide two ways to instantiate O-TTBE. First, we present a concretely efficient instantiation of O-TTBE from bilinear pairings in the random oracle model. Our construction uses bilinear pairings to augment the asymmetric ElGamal encryption system [97] with an oblivious tag. In ElGamal, the encryption of a message m is computed as $(A = g^a, M = m \oplus H(Y^a))$ for a freshly sampled exponent a , a group generator g , a random oracle H , and a public key $Y = g^y$. In the threshold setting, the secret key y is shared among the committee members; each member receives a share y_i . For decryption, the servers compute decryption shares $D_i = A^{y_i}$. We add the tag dependency to the ElGamal scheme by applying a random offset T to the encryption randomness A in both encryption and decryption. The offset T is sampled by a random oracle with input tag t and encryption randomness A . This gives us a unique offset T for each tag-ciphertext pair. Moreover, the random oracle ensures that each offset is uniformly random and sampled independently from any other offset. The offset cannot simply be applied using addition or multiplication, e.g., $M = H(Y^{a \cdot T}) \oplus m$, as this allows an adversary to exploit homomorphisms between decryptions with different tags,

4. Statement Oblivious Witness Encryption

i.e., $Y^{a \cdot T} = (A^{x \cdot T'})^{\frac{T}{T'}}$. To avoid such attacks, we apply the offset using a bilinear pairing e , i.e., $M = m \oplus H(e(T, X^a))$. During decryption, the tag is applied before the secret key operation, i.e., $D_i = e(T, A)^{y_i}$. This way, an invalid tag is ensured to apply a uniform random offset to the decryption. We further prevent the reuse of encryption randomness A in new ciphertexts by appending a zero-knowledge proof of knowledge of a to the ciphertext and binding the proof to the message-dependent component M .

Second, we provide an instantiation of O-TTBE from collision-resistant hash functions, one-time signatures, and anonymous threshold identity-based encryption (A-TIBE). This transformation follows the approach of [32], which uses the same building blocks, just without the anonymity property, to construct CCA secure threshold encryption. In a nutshell, our construction works as follows. A party, Alice, encrypts a message in the O-TTBE scheme by first sampling a fresh one-time signature key pair and creating a new identity based on the tag and the new signature verification key. Then, she computes an A-TIBE ciphertext of the message under the freshly generated identity. In order to bind the ciphertext to the generated identity, Alice signs the ciphertext and the signature verification key with the corresponding signing key and appends the signature to the ciphertext. The anonymity property of the A-TIBE scheme ensures that the tag remains private. For decryption, the servers generate the identity key for the identity determined by the given tag and the signature verification key embedded in the ciphertext. Any decryption attempt with an incorrect tag will yield a new identity such that the generated identity key does not leak any information about the plaintext or the tag used for encryption.

While there are constructions for anonymous identity-based encryption [35, 100] and threshold identity-based encryption [32], we are the first to consider the notion of anonymous threshold identity-based encryption. To provide a first step towards instantiating A-TIBE, we show how A-TIBE can be instantiated from anonymous non-threshold identity-based encryption and homomorphic secret sharing (HSS). While Campanelli et al. [49] have already used HSS for a similar transformation, just without the anonymity property, we discovered a gap in their security analysis. Their construction does not provide any means to verify if an identity key has been generated correctly, a feature that is required by threshold identity-based encryption schemes. To address this problem, we augment the non-threshold anonymous identity-based encryption scheme with a verifiability property that allows to check if an identity key has been generated correctly. In our transformation, the A-TIBE scheme inherits this property from the underlying non-threshold scheme. We note that our construction relies on general-purpose HSS and, hence, is not concretely efficient yet. However, our transformations use the underlying primitives in a

4. Statement Oblivious Witness Encryption

black-box way such that every progress in the field of A-TIBE or HSS directly translates to SO-TWE.

4.3. Related Work

Goyal et al. [106] are the first to propose the utilization of a committee to provide functionality equivalent to witness encryption. Their notion, called *extractable Witness Encryption on Blockchains (eWEB)*, is explicitly tailored to the blockchain setting by relying on a blockchain election mechanism to provide committees. Furthermore, eWEB explicitly integrates a mechanism that allows committee renewal, i.e., the committee can be securely replaced by a new committee that receives all the secrets held by the old one. As committee renewals aim to guard against adversaries who corrupt additional committee members over time, Goyal et al. prove the security of their construction in the presence of an adaptive adversary. On the downside, their approach requires servers to keep shares of all secrets in storage and even transfer them upon committee renewal. Further, they do not consider the privacy of statements used during encryption. Our scheme, in contrast, abstracts over the origin of the committee, allowing compatibility with different committee election mechanisms. As the committee, in our notion of threshold witness encryption, decrypts received ciphertexts upon requests, we require the committee to store nothing but a constant size secret key share. Unlike [106], we do not explicitly consider committee renewals and, hence, focus on the weaker static security setting. However, in our concrete ElGamal-based instantiation, the committee members hold Shamir secret shares of a single field element. Therefore, it is fair to assume that the committees can be renewed with minimal overhead. Finally, our constructions are tailored to support the privacy of statements used for encryption.

Subsequent to [106], Campanelli et al. [49] propose another construction for blockchain-based witness encryption called *Blockchain Witness Encryption*. However, their construction is neither practical, e.g., each encryption requires the deployment of a smart contract, nor does it consider statement privacy.

5. Virtual Trusted Third Parties

Many problems studied in cryptography would become trivial if there were a trusted third party (TTP). This observation is illustrated by the fact that simulation-based security defines the security of a protocol with respect to an ideal world in which a TTP exists. While we are not in an ideal world and, hence, cannot assume the existence of a real TTP, it is possible to design systems emulating some of the features provided by a TTP. We call such a system a *virtual trusted third party (V-TTP)*. Ideally, a V-TTP strictly adheres to the expected behavior, keeps its state and communication confidential, is capable of performing complex computations, and provides high availability guarantees.

One approach for instantiating a V-TTP is to rely on a committee of servers that stores the internal state of the V-TTP in a secret shared way and performs computations via secure multiparty computation (MPC) as, for example, done in [22, 31, 53, 121, 138]. This approach ensures correct execution of the V-TTP and confidentiality of state and messages but suffers from a significant performance overhead introduced by the underlying MPC protocol. Furthermore, there is an inherent trade-off between availability and security; the higher the availability guarantees, the less malicious parties can be tolerated within the committee. An alternative approach is to rely on trusted execution environments (TEEs), e.g., [17, 109, 129, 149]. TEEs ensure correct execution of the installed code, guard the internal state and communication, and are capable of executing complex applications. However, TEEs are in control of potentially untrusted operators, which can censor communication and, hence, harm the availability of the service.

In order to overcome the availability restrictions of the above two approaches, one can rely on blockchains, which provide high availability guarantees by design. The naive approach is to realize the V-TTP in the form of a smart contract running directly on the blockchain. Smart contracts ensure the correct execution of the deployed code and inherit high availability guarantees from the underlying blockchain. However, traditional smart contracts are inherently public and limited in the extent of computations they can perform. The latter makes computations assigned to smart contracts costly and, hence, inefficient. While there are approaches based on MPC techniques or TEEs that enhance traditional smart contracts with additional privacy guarantees or relieve smart contracts from their

5. Virtual Trusted Third Parties

complexity limitations, e.g. [56, 73, 79, 80, 120, 128, 142], all known solutions suffer from at least one of the following shortcomings: They require participants to lock large amounts of collateral used to incentivize availability; they incorporate regular blockchain interactions, which is costly; they restrict the scope to applications with a limited lifetime or a fixed set of participants; or they do not hide the private state of the smart contract from all of the involved parties.

5.1. Contribution

In this thesis, we propose POSE a novel blockchain-based system that leverages a committee of TEE operators to execute smart contracts with a private state and significantly increased efficiency compared to traditional smart contracts without suffering from any of the shortcomings introduced above. Our solution circumvents these shortcomings by making a minor sacrifice in terms of availability, i.e., POSE only guarantees the availability of a smart contract as long as at least one TEE operator in a randomly selected pool of operators remains responsive. Our work has been disseminated in the following article and can be found in Appendix E.

- [95] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A. Sadeghi. “POSE: Practical Off-chain Smart Contract Execution”. In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. 2023. **Part of this thesis.**

5.2. Key Results

In the following, we present the high-level design and mechanics of the POSE system. We refer the reader to our published article [95] (Appendix E) for further details.

System overview. The POSE system assigns each POSE contract a randomly sampled pool of TEE operators that are jointly responsible for managing the contract, which in turn can be used to implement a V-TTP. The system only relies on the blockchain to resolve disputes. In the optimistic case, POSE contracts are executed without any blockchain interaction. Furthermore, POSE guarantees the availability of a contract as long as at least one of the operators is honest, i.e., remains active.

We illustrate the high-level overview of the POSE system in Figure 5.1. The core of the system is a smart contract deployed to the blockchain, called the

5. Virtual Trusted Third Parties

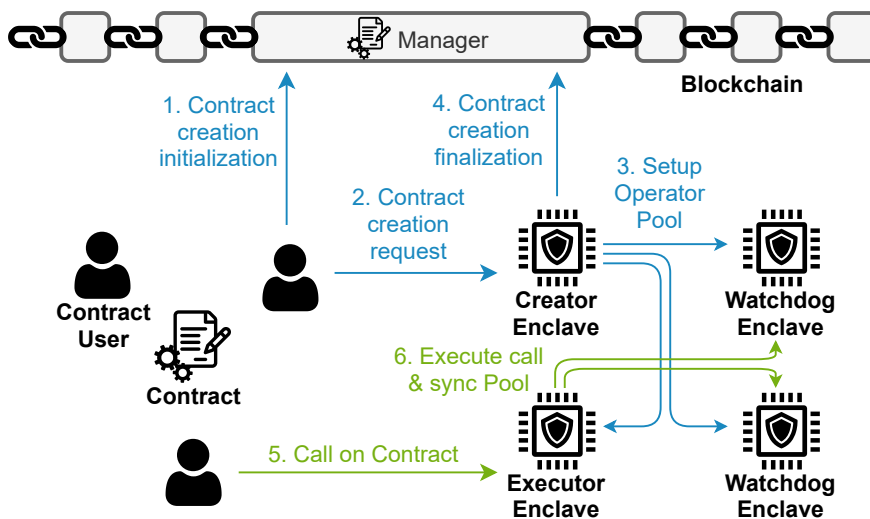


Figure 5.1.: Overview of the POSE system including the processes of contract creation (blue) and the contract execution (green).

manager. The manager keeps track of all registered POSE enclaves and POSE contracts and resolves disputes when necessary. TEE operators who wish to join the system install a new enclave running a special POSE program on their TEE. The enclave generates a new key pair and returns its public key and an attestation report to the operator. The operator then registers the TEE with the manager by submitting the attestation report and the generated public key. The manager utilizes the attestation report to verify that the enclave runs the correct program and includes the enclave in its list of available enclaves. The uploaded public key is employed to encrypt and authenticate communication with the enclave, thereby preventing the operator from eavesdropping or tampering with the communication of its enclave. In the following, we will assume that all communication involving enclaves is protected in this manner. However, operators may still drop, delay, or replay messages.

In order to deploy a new POSE contract, a client selects a random POSE enclave, designated as the *creation enclave*, from the manager’s list and sends a creation request to the manager containing the identity of the creator enclave (Step 1). Next, the client sends the contract code to the creator enclave (Step 2). The creator enclave selects a random pool of enclaves from the manager’s list, samples shared contract secrets, such as a shared symmetric encryption key, and sends the code and the initial secrets to the selected enclaves (Step 3). The selected enclaves install the new contract with the given code and the provided secrets and confirm the installation to the creation enclave. The state of the installed contract contains

5. Virtual Trusted Third Parties

both the application-specific variables and the variables required to manage the contract in the context of the POSE system. As the state is only stored within protected enclaves, the operators cannot access the state of POSE enclaves. Once the creator enclave receives installation confirmations from all pool members, it sends a finalization report containing the identities of the pool members to the manager (Step 4).

During contract execution, pool members can have two different roles. The first enclave in a pool is the *executor enclave*. The other pool members are *watchdogs*. A client executes a contract by sending an execution request to the executor enclave (Step 5). The executor enclave executes the request and distributes the resulting state to the watchdogs, who confirm the state update (Step 6). Finally, the executor enclave sends an execution confirmation, including potential outputs of the transaction, to the client. Optimistically, i.e., if all actors are responsive, the execution process does not involve any communication with the blockchain.

Handling unresponsive enclaves. The rationale behind the pooled execution is that unresponsive executors can be kicked from the pool and replaced by one of the watchdogs. This way, POSE ensures the availability of a contract as long as at least one of the operators in the contract’s pool is honest, i.e., remains active and responsive. However, for watchdogs to be able to replace the executor, it is necessary to ensure that each watchdog receives all state updates. Therefore, we only allow an executor enclave to finalize a transaction if it receives confirmations from all watchdogs. In order to prevent unresponsive watchdogs from stalling the execution, we enable the executor enclave to kick unresponsive watchdogs and continue the execution with a smaller pool.

As operators can censor their enclaves’ communication, it is impossible for any enclave to determine the unresponsiveness of other enclaves on its own. An executor enclave missing an update confirmation cannot differentiate whether its own operator blocks the communication or the watchdog enclave failed to send the confirmation. Watchdog enclaves, on the other hand, cannot know if an executor that should be kicked is indeed unresponsive or if a client only claims so. In the blockchain space, this problem is known as *non-uniquely attributable faults*: malicious behavior is evident but cannot be attributed to a specific actor.

In order to solve this problem, we utilize the manager to resolve disputes about the unresponsiveness of enclaves. As the manager is executed on the blockchain, its state and communication are accessible to all parties such that messages sent to the manager are evident to all actors. Whenever an enclave stops responding at any point throughout the protocol, its operator is challenged by the dependent party via the manager. The dependent party can be a client waiting for a con-

5. Virtual Trusted Third Parties

firmation of a creator enclave or an executor enclave, the creator enclave waiting for an installation confirmation of a pool member, or an executor enclave waiting for an update confirmation of a watchdog. To issue the challenge, the dependent party submits the identity of the challenged enclave and the (encrypted) message, that should be answered, to the manager. The operator of the challenged enclave answers the challenge by submitting its enclave's response to the manager. This way, the dependent party receives the requested response and, hence, is able to continue with the execution. If the challenged operator fails to respond timely, its enclave is kicked from the pool and unregistered with the manager. As creator enclaves have no immediate replacement, clients have to restart the creation procedure if a creator enclave does not respond to an issued challenge.

Supplying enclaves with blockchain data. Some of the actions taken by an enclave depend on data published on the blockchain, such as challenges or responses sent to the manager. Therefore, it is essential to ensure consistency of the blockchain data available to an enclave. As an enclave does not have direct access to the blockchain network, it depends on the data provided by its operator. However, an operator can be malicious and, therefore, provide inconsistent or incomplete data. Consequently, we have designed a synchronization mechanism that ensures the correctness and completeness of the blockchain data forwarded to the enclave, limits the time an enclave may lag behind the main chain, and prevents the isolation of enclaves onto maliciously crafted forks. Our synchronization mechanism is tailored to proof-of-work blockchains. If the POSE system is instantiated on top of proof-of-stake-based blockchains, the synchronization needs to be adapted to the respective consensus protocol.

Upon enclave initialization, the enclave receives the most recent finalized block and adds the block hash to its attestation report, which is submitted to the manager upon registration. The manager uses this hash to reject registrations of enclaves that are not up to date. More precisely, the manager rejects the registration if the provided hash does not match any of the Δ most recent blocks, where Δ defines the upper limit for the time an enclave is allowed to lag behind the main chain. Once initialized, the enclave synchronizes itself with the blockchain as follows: Upon execution, the enclave receives the blocks mined since its last execution and validates that the new blocks are valid successors of its own sub-chain. To prevent an operator from isolating the enclave onto a fork, we require the operators to periodically provide new blocks to the enclave even if the enclave is not required to perform any actions. If the enclave does not receive sufficiently many blocks in a fixed time interval, it shuts itself down. The time interval and the number of expected blocks are defined based on the blockchain's average block

creation time, variances in the block creation time, and the speed of the difficulty adjustment.

5.3. Related Work

V-TTP systems relying solely on MPC or TEEs have already been introduced in the motivation above. In the following, we will focus on smart contract-based approaches proposing platforms that enhance traditional smart contracts, already providing high availability guarantees and correct execution, with a private state or increased efficiency.

State channels [79, 80, 142] release smart contracts from their complexity limitations by outsourcing the execution of the contract to a protocol executed between the channel members. The blockchain is only invoked to resolve disputes or capture results of the channel execution, e.g., to disburse coins assigned to the channel. While reliably addressing the complexity limitations of smart contracts, state channels require participants to lock collateral, do not support a private smart contract state, and are limited to applications with a fixed group of participants.

Rollups, studied both in industry [7, 148, 162, 183] and academia [120, 174, 175], increase the efficiency of smart contracts by outsourcing computations to an operator or a committee of operators. Operators compute (batches of) transactions and submit the result of the outsourced computation to the blockchain. While relieving the blockchain from the necessity of computing the transaction itself, the state of the contracts is still stored on the blockchain. Rollups support smart contracts with an unlimited lifetime and open participation but still require regular blockchain interactions and do not support contracts with a private state. One partial exception is [120], which stores only the hash of the contract state on the blockchain, thereby minimizing the complexity of the regular blockchain interaction and providing state privacy if the operators are honest. However, [120] privacy of the contract state can only be guaranteed if all operators assigned to a contract are honest.

Steffen et al. [163, 164] enhance the privacy properties of smart contracts by relying on zero-knowledge proof systems and asymmetric encryption. In their system, contract variables can explicitly be declared private. Each private variable is assigned to a particular owner. Instead of storing the private variables in clear, the contract stores the ciphertext of the private variables encrypted under the respective owner's public key. In order to invoke a function, clients locally compute the original function operating on the plaintext variables and submit the state updates, encrypted in the case of private variables, to the smart contract together with a

5. *Virtual Trusted Third Parties*

zero-knowledge proof showing that the function has been computed correctly. As each variable has a dedicated owner, it is not possible to compute functions that involve variables of multiple owners without first disclosing the private variables to the party executing the function. This shortcoming has been partially addressed in [163] by utilizing additively homomorphic encryption, which allows parties to perform additions on private variables of different owners. Nevertheless, neither system supports variables that are kept private from all parties nor addresses the complexity limitations of smart contracts.

Another line of work [19, 20, 21, 128, 184] utilizes MPC in order to realize smart contracts with a private state. Computations and storage are outsourced to a committee of MPC nodes that securely stores the private state in a secret shared way and securely performs computations on the state via multiparty computation. If the smart contract execution yields results that should be captured by the blockchain, e.g., the payout of coins, the consortium submits the result to the blockchain, together with a proof showing that the contract has been executed correctly. While the MPC-based approach supports smart contracts with a private state, unlimited lifetime, and open participation without requiring regular blockchain interactions, the MPC protocol introduces an additional runtime overhead. Furthermore, the secret sharing of the state and the computation comes with an inherent trade-off between availability and security; the more unresponsive parties can be tolerated, the fewer malicious parties are required to break security. To address this trade-off, responsiveness is often incentivized via collateral deposited by the committee members, which is retained in case of unresponsiveness.

Ekiden [56] utilizes TEEs to support smart contracts with a private state. In Ekiden, the state of each contract is stored in an encrypted form on the blockchain. The decryption keys are maintained by a TEE-enabled key management committee. Contract execution is facilitated by operators running special computation enclaves on their TEEs. To execute a contract, a client submits an execution request to any computation enclave in the system. The computation enclave retrieves the encrypted contract state from the blockchain, requests the corresponding decryption key from the key management committee, decrypts the contract state, executes the requested transactions, re-encrypts the new state, stores the encrypted new state on the blockchain, and triggers public effects of the state transition, e.g., the payout of coins. While Ekiden provides high availability guarantees and supports contracts with a private state, each contract invocation requires costly interaction with the blockchain.

FastKitten [73] is another TEE-based solution that supports smart contracts with a private state. In FastKitten, contracts are deployed directly to the enclave of an operator and executed by the enclave. To prevent an operator from shutting

5. *Virtual Trusted Third Parties*

down the enclave, potentially freezing coins of the deployed smart contract, the operator has to lock collateral, which is used to reimburse participants for potential losses if the enclave becomes unavailable. Analogous to state channels, the blockchain is only invoked to settle disputes or capture the public effects of the contract execution. While FastKitten addresses both the complexity limitations of smart contracts and the missing privacy guarantees for the contract state, the collateral restricts the platform to contracts with a limited lifetime and a fixed set of participants.

Unlike previous work, POSE provides a platform to execute smart contracts and, hence, instantiate V-TTPs that addresses all of the aforementioned shortcomings in one holistic solution. More precisely, POSE neither requires locked collateral nor regular blockchain interactions, supports complex contracts with open participation and unlimited lifetime, and hides the state of smart contracts from all involved parties. On the downside, POSE makes a minor sacrifice in terms of availability when compared to traditional smart contract platforms executing contracts directly on the blockchain. While traditional smart contracts are available as long as the underlying blockchain system remains healthy, POSE requires, in addition to a healthy blockchain system, at least one TEE operator in a randomly selected pool to remain responsive.

6. Modern Programming Language for Decentralized Applications¹

Today, the contracts and clients that make up a decentralized application (dApp) are implemented as separate programs in different programming languages. Contracts have a set of public functions that can be called by clients. By restricting the states from which certain functions can be called, programmers implicitly define a finite state machine (FSM) that describes the program flow of the dApp. Figure 6.1 shows a simplified Solidity² smart contract for a tic-tac-toe dApp with a dedicated funding phase and payout phase and the corresponding FSM.

```
1 contract TicTacToe {
2   State phase = Funding; /*...*/
3   function Fund() public {
4     require(phase == Funding);
5     /*...*/;
6     if (!(balance < FUNDING_GOAL))
7       phase = Exec; }
8   function Move(int x, int y) public {
9     require(phase == Exec && sender == players[moves % 2]);
10    /*...*/;
11    if (!(moves < 9 && winner == 0))
12      phase = Finished; }
13   function Payout() public {
14     require(phase == Finished);
15     /*...*/;
16     phase = Closed; }
17 }
```

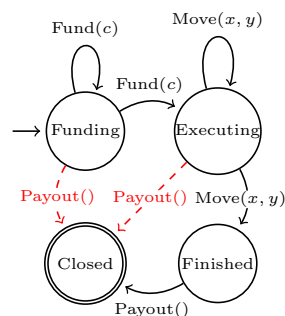


Figure 6.1.: Solidity code of a tic-tac-toe contract with a funding phase and a payout phase (left). The code implicitly defines a finite state machine (right). Red dashed arrows illustrate unintended transitions, e.g., premature calls of the Payout function. Line 14 of the contract (underlined in red) prevents such unintended payouts.

¹Figures and listings are taken from [152] with minor adjustments.

²Solidity is the most widely used programming language for smart contracts executed on the Ethereum Virtual Machine (EVM). The EVM is the most popular smart contract platform used not only by Ethereum [83] but also by other smart contract capable blockchains such as [15, 29, 166].

6. Modern Programming Language for Decentralized Applications

During the initial **Funding** state, the two players jointly deposit coins until a certain funding goal is reached. Then the contract enters an **Executing** state, in which the players take turns submitting their moves. When the game is over, the contract enters the **Finished** state, where either player can trigger a payout. Once the payout has been triggered, the contract terminates by reaching the **Closed** state. The programmer specifies the states in which each function can be executed by placing entry guards. In this way, the programmer protects the program flow from unintended deviations. For example, a premature call to the **Payout** function could deprive a player of its well-deserved winnings. Therefore, it is essential to ensure that the **Payout** function can only be executed after the application has reached the **Finished** state.

The FSM-style encoding of the program flow has several drawbacks. First, it makes the distributed program flow difficult to express and reason about. Second, it requires the programmer to explicitly protect each function against unintended invocation, e.g., to validate both the caller and the current state upon invocation. This increases the risk of vulnerabilities due to mismanaged program flow. As smart contracts directly control financial assets and are publicly available, mismanaged program flow can cause tremendous damage. Real-world examples of exploits due to mismanaged program flow include the DAO hack, where attackers stole \$50 M USD [5], and the two Parity hacks, where attackers stole \$30 M USD [4] and froze \$150 M USD [4]. Therefore, we believe it is essential, especially for the use of smart contracts in cryptographic protocols, to develop a simpler approach to dApp development that makes it easier to express distributed program flow and automatically protects the specified program flow from unintended client deviations.

6.1. Our Contribution

In this thesis, we present **Prisma**, a modern programming language for decentralized applications. In **Prisma**, we interpret the smart contract as the active entity that is in charge of the control flow and passes it to the client when input is required. This allows programmers to explicitly define the distributed program flow of the decentralized application using standard control flow instructions. **Prisma** automatically protects the program flow against client-sided deviations. Furthermore, **Prisma** allows the implementation of both contract and client in one unit in the same programming language and to express communication via explicit client calls, thus simplifying the encoding of client-contract communication and rendering mismatching communication impossible. Our work has been disseminated in

the following article and can be found in Appendix F.

- [152] D. Richter, D. Kretzler, P. Weisenburger, G. Salvaneschi, S. Faust, and M. Mezini. “Prisma : A Tierless Language for Enforcing Contract-client Protocols in Decentralized Applications”. In: *ACM Trans. Program. Lang. Syst.* 3 (2023), 17:1–17:41. **Part of this thesis.**

6.2. Key Results

In the following, we present the high-level concept of Prisma and parts of our experimental evaluation showing that the additional layer of abstraction introduced by Prisma causes only a tolerable performance overhead. We refer the reader to our published article [152] (Appendix F) for further details.

The programming language. Prisma is implemented as a domain-specific language integrated with Scala. We provide a compiler that compiles Prisma code into Solidity contract code to be deployed on a blockchain and Scala client code to be executed by users. Below, we explain the syntax of Prisma using the code example in Figure 6.2. The given program shows a Prisma implementation, defining both client and contract, of the tic-tac-toe application introduced above (cf. Figure 6.1).

```

1  @prisma object TicTacToeModule {
2
3  @co @cl case class UU(x: U8, y: U8)
4
5  class TicTacToe(
6    val players: Arr[Address],
7    val fundingGoal: Uint) {
8
9    // u8 is an unsigned 8-bit integer
10   @co @cross var moves: U8 = "0".u8
11   @co @cross var winner: U8 = "0".u8
12   @co @cross val board: Arr[Arr[U8]] =
13     Arr.ofDim("3".u, "3".u)
14
15   @co def performMove(x: U8, y: U8): Unit =
16     { /* ... */ }
17   @cl def updateBoard(): Unit =
18     { /* ... */ }
19   @cl def fund(): (U256, Unit) =
20     (readLine("How much?").u, ())
21   @cl def move(): (U256, UU) =
22     ("0".u, UU(readLine("x-pos?"),
23               readLine("y-pos?")))
24
25   @cl def payout(): (U256, Unit) = {
26     readLine("Press (enter) for payout")
27     ("0".u, ())
28   }
29   @co val init: Unit = {
30     while (balance() < FUNDING_GOAL) {
31       awaitCl(_ => true) { fund() }
32     }
33     while (moves < "9".u && winner == "0".u) {
34       val pair: UU = awaitCl(a =>
35         a == players(moves % "2".u)) { move() }
36       performMove(pair.x, pair.y)
37     }
38     awaitCl(a => true) { payout() }
39     if (winner != "0".u) {
40       players(winner - "1".u).transfer(balance())
41     } else {
42       players("0".u).transfer(balance() / "2".u)
43       players("1".u).transfer(balance())
44     }
45   }
46 }

```

Figure 6.2.: Prisma dApp implementing the tic-tac-toe case study.

6. Modern Programming Language for Decentralized Applications

In *Prisma*, programmers place variable or function declarations on the contract or client using the annotations `@co` and `@cl`. The variable declarations can be annotated with an additional `@cross` (cf. Line 10) to grant the client read access to the contract variable. Data types passed between contract and client are defined as classes and annotated with both `@co` and `@cl` (cf. Line 3).

The program flow is defined by the `init` block placed at the contract. This block provides the functionality of a main function known from other programming languages. When the contract is deployed, the program flow starts at the beginning of the `init` block. Code is executed on the contract until an `awaitCl`-expression is encountered. This expression passes control to the clients, prompting them to submit input in the form of a message. The `awaitCl`-expression takes two inputs. The first is a lambda expression, which receives a client's address as input and returns a boolean. Clients execute the expression with their own address as input to decide whether it is their turn to send a message. The contract uses the same expression to decide whether to accept a message received from a client. In this way, we force developers to explicitly specify access control, thereby reducing the risk of human failure. The second input to the `awaitCl` expression is a code block executed by the client. It returns the amount of coins transferred to the contract and a message. The message is passed to the contract by the `awaitCl` expression. The amount of Ether transferred can be accessed through a built-in expression `value`. In order to access other blockchain-specific functionality, we provide additional built-in expressions, such as `balance` (cf. Line 29) to access the current balance of the contract, or `transfer` (cf. Line 39) to send Ether to another address.

Consider the `init` block in the given code example (Lines 28-44) for illustration. The contract first requests clients to provide funding via the `awaitCl`-expression in Line 30. Since any client can provide funding, the access control always returns `true`. Upon receiving the control, the clients execute the `fund` function, which requests a command line input from the user specifying the funding amount and sends the funding along with an empty message to the contract. This step is repeated until a specific funding goal is reached (cf. Line 29). The contract then requests the two players to submit their moves in alternating turns until one party wins or the board is full (cf. Line 32 and 33). Once a move has been received, the board is updated accordingly and the winning condition is checked. Since only one player may submit a move in each round, the `awaitCl`-expression in Line 33 validates the sender of the message. Once the game ends with a winner or a tie, either client can request a payout by sending a message with neither coins nor content (cf. Line 37). Upon receiving the payout request, the contract pays out the funds accordingly.

Efficiency evaluation. In order to evaluate the overhead of the additional layer of abstraction introduced by **Prisma**, we implemented 12 case studies in **Prisma** and Solidity and compared the gas consumption. Since our compiler transforms **Prisma** code into Solidity code, it is not possible for **Prisma** to outperform hand-written Solidity code; any smart contract compiled from **Prisma** can also be implemented directly in Solidity. Nevertheless, the results of our evaluation show that the contract deployment overhead is 0-7% and the contract execution overhead is 0-10% for all case studies.

The overhead of **Prisma** in some of the case studies is primarily attributable to the automated program flow control. In order to ensure correct execution, **Prisma** introduces an explicit phase variable upon compilation. The phase variable is used to generate entry guards for each of the contract functions, analogous to those utilized in the Solidity code example in Figure 6.1 (Lines 4, 9 and 14). In Solidity, developers do not always need an explicit phase variable. In some cases, they can infer the phase from existing variables. For example, in the tic-tac-toe game, the phase information can be inferred from the contract balance, the move variable, and the winner variable. Case studies that require an explicitly managed phase variable, also when implemented directly in Solidity, exhibit only a negligible overhead.

6.3. Related Work

The programming languages most closely related to **Prisma** are Solidity [161], Obsidian [58, 59, 60], and Nomos [72]. While the former represents the de facto standard for EVM-based smart contracts, the latter two are academic programming languages for smart contracts enforcing correct execution of the intended distributed program flow by design.

Solidity has the aforementioned shortcomings. First, contracts and clients are implemented separately in different programming languages. Second, the intended program flow needs to be implicitly encoded as a finite state machine. Third, the program flow needs to be guarded manually against both out-of-order and unauthorized execution of functions. In contrast, **Prisma** enables the programmer to implement both the contract and the client in a single unit, thus eliminating the possibility of mismatching communication. Furthermore, **Prisma** allows for the explicit specification of the intended program flow, automatically guards the application against client-side deviations, and forces the developer to explicitly define access control.

Obsidian is a programming language for smart contracts that uses so-called

6. Modern Programming Language for Decentralized Applications

typestates [74, 165] to increase safety by forcing developers to explicitly define the finite state machine representing the intended program flow. As illustrated in the Obsidian code snippet in Figure 6.3, the Obsidian code explicitly defines a set of valid states and the allowed transitions between the states.

```

1  asset contract TTT {
2  state Funding{}; state Executing{}; state Finished{}; state Closed{}
3  transaction Fund(TTT@Funding>>(Funding|Executing) this, int c) {
4    /*...*/; if (/* enough funds? */) -> Executing else -> Funding }
5  transaction Move(TTT@Executing>>(Executing|Finished) this, int x, int y) {
6    /*...*/; if (/* game over? */) -> Finished else -> Executing }
7  transaction Payout(TTT@Finished>>Closed this) {
8    /*...*/; -> Closed } }

```

Figure 6.3.: Simplified Obsidian smart contract of the tic-tac-toe case study.

Nomos is a smart contract programming language that employs so-called *session types* [47, 48, 114, 115, 170] to protect the program flow. As illustrated in the code snippet in Figure 6.4, the Nomos code defines a set of valid states, the inputs that can be provided in each state and the possible state transitions (cf. Lines 1-3). The final state is always encoded as 1. The transition logic is implemented via processes (cf. Lines 4 and 8). The `recv`-expression accepts a session type of the form $T \rightarrow U$ as input, returns a variable of type T , which represents the client message, and transitions the session to type U . If a type has multiple possible transitions, the program needs to select the respective label (e.g., `$ s.notenough` in Line 6 or `$ s.enough` in Line 7). Note that the payout does not require an additional process, as it has only one entry point and, therefore, can be appended to the transition to the `Finished` state (cf. Line 11).

```

1  type Funding = money -> +{ notenough: Funding, enough: Executing }
2  type Executing = int -> int -> +{ notdone: Executing, done: Finished }
3  type Finished = _ -> 1
4  proc contract funding : . |{*}- ($ s : Funding) = {
5    a = recv $ s; /* ... */
6    if /* enough funds? */ then $ s.notenough; $ s <- funding
7      else $ s.enough; $ s <- executing }
8  proc contract executing : . |{*}- ($ s : Executing) = {
9    x = recv $ s; y = recv $ s; /* ... */
10   if /* game over? */ then $ s.notdone; $ s <- executing
11   else $ s.done; z = recv $ s; /* distribute funds */ close $ s }

```

Figure 6.4.: Simplified Nomos smart contract of the tic-tac-toe case study.

In contrast to both Obsidian and Nomos, Prisma relies on a standard type system and employs standard expressions to specify the distributed program flow.

6. Modern Programming Language for Decentralized Applications

The sole non-standard expression introduced by **Prisma** is the **awaitCl**-expression. However, this expression functions in a manner analogous to the **await**-expression commonly utilized in other programming languages that facilitate asynchronous interactions, such as JavaScript [81]. Consequently, it adheres to a well-established semantic framework. This leads to a much simpler and clearer specification of the distributed program flow in **Prisma** in comparison to the alternative smart contract languages, as illustrated by the provided code snippets. Furthermore, neither Obisidian nor Nomos enables the implementation of client and contract in one unit in the same programming language as possible in **Prisma**.

7. Conclusion

Today’s blockchain systems offer a wide range of features going far beyond the mere execution of financial transactions. Recently, a new trend in cryptography emerged that leverages the rich set of features provided by modern blockchain systems to implement new cryptographic services and enhance existing ones. However, the potential of blockchain systems is far from exhausted. This thesis, therefore, aimed to unlock further potential of blockchain systems and to provide more advanced blockchain-based cryptographic services by identifying and addressing gaps in prior work on blockchain-based cryptography. First, we have shown how to augment arbitrary semi-honest secure protocols with covert security and smart contract-based punishment of detected cheating attempts, thereby increasing the deterrent effect of cheating detection. Second, we have enhanced blockchain-based witness encryption, as introduced by [106], with additional privacy guarantees for the statement used for encryption. Third, we have proposed a novel smart contract platform to provide a service that effectively emulates a trusted third party and, unlike previous approaches, combines increased efficiency and state privacy with high liveness guarantees. Fourth, we have proposed a new programming language for decentralized applications that allows developers to implement blockchain-based applications in a simpler and more secure way. We conclude the thesis by discussing further research questions based on our work.

The concrete efficiency gain of covert security. The primary motivation for the utilization of covert security is that covert security constitutes a middle-ground between active and passive security, providing higher security guarantees than passive security and more efficient protocols than active security. However, continuous research on actively secure multiparty computation, e.g., [25, 38, 54, 71, 76, 112, 146, 171], is gradually narrowing the gap between passive and active security. In particular, the recently introduced concept of pseudorandom correlation generators (PCG) [36, 37, 38, 39, 64, 76, 156, 179] is assumed to provide significant efficiency improvements.¹ PCGs allow parties to generate the precomputation

¹PCGs still lack complete implementations, making it difficult to quantify the concrete runtime benefits.

7. Conclusion

material required by MPC protocols in the offline/online model with sublinear communication complexity; sublinear in the amount of required precomputation material and, hence, in the complexity of the jointly computed function. While PCGs can be used for both active and passive security, they have a much more substantial impact on active security.

Given the recent advancements in active security, we believe it is an interesting research question to empirically examine the concrete efficiency advantage of covert security over active security. In both active security and covert security, there are a variety of protocols that focus on different settings, e.g., two-party and multiparty protocols, and offer different trade-offs, e.g., communication complexity versus computational complexity. Therefore, it is likely that the benefit of covert security strongly depends on the concrete setting. A comprehensive experimental evaluation and comparison of the various protocols across different settings could provide a more thorough basis, allowing system designers to choose the best protocol for their particular setting and to assess whether a downgrade from active to covert security is worth the efficiency gain.

Anonymous threshold identity-based encryption. One of our constructions for statement oblivious threshold witness encryption is based on anonymous threshold identity-based encryption. While there are constructions for anonymous identity-based encryption [35, 100] and threshold identity-based encryption [32], there has not been any prior work combining both anonymity and threshold identity key issuance. In [91], we provide an instantiation for anonymous threshold identity-based encryption based on general-purpose homomorphic secret sharing. Since there are no efficient instantiations of general-purpose homomorphic secret sharing, this instantiation represents a feasibility result.

Therefore, we consider it to be an interesting direction for future work to construct concretely efficient anonymous threshold identity-based encryption schemes. Such constructions would not only benefit our work on statement oblivious threshold witness encryption but could also enhance use cases of anonymous non-threshold identity-based encryption with threshold identity key issuance, thus mitigating the single point of failure represented by the master key holder.

POSE on proof-of-stake blockchains. Although most of the POSE system is independent of the underlying blockchain, we have based POSE on the Ethereum blockchain [83], the most prominent smart contract platform. Most importantly, we have designed the blockchain synchronization with respect to the Ethereum consensus protocol. At the time the POSE protocol was developed, Ethereum’s con-

7. Conclusion

sensus protocol was based on proof-of-work. However, in the meantime, Ethereum has shifted to proof-of-stake [82]. In fact, most of the actively used smart contract platforms are based on proof-of-stake blockchains [2, 15, 29, 52, 160, 166].

Therefore, we believe it is an interesting research direction for future work to design alternative synchronization techniques that enable enclaves to ensure the correctness and completeness of the received blockchain data on proof-of-stake blockchains. Most importantly, it is necessary to prevent a malicious operator from isolating its enclave on a blockchain fork, on which its enclave is the only member of a pool responsible for a particular POSE contract. Such an attack would allow the operator to test a particular transaction on its isolated enclave, obtain the result, which may depend on the private state of the contract and, therefore, cannot be determined without isolation, and, if beneficial, submit the transaction to the original pool running with respect to the main chain.

Eliminating inefficiencies in the Prisma compiler. Our evaluations of the Prisma compiler show that for some case studies, Prisma-compiled smart contracts introduce an overhead of up to 7 % in deployment costs and up to 10 % in execution costs compared to equivalent smart contracts implemented in Solidity. This overhead is due to the automated program flow protection provided by Prisma. Prisma contracts implicitly introduce a phase variable to keep track of the application’s phase and use this variable to protect the program flow by rejecting unintended function calls. However, in some applications, it is possible to infer the application’s phase from the existing contract variables, making the phase variable redundant and, hence, causing the efficiency overhead.

An interesting direction for future work is to enable the Prisma compiler to determine whether a phase variable is necessary. The compiler could deploy program flow analysis techniques such as symbolic execution [124] to determine whether the application’s phase is uniquely defined by the existing contract variables throughout the lifetime of the dApp. If this is the case, the compiler generates entry guards for functions based on the rules derived from the program flow analysis instead of using a dedicated phase variable. Such an extension would not only eliminate the inefficiencies introduced by the Prisma compiler, but is likely to detect the redundancy of the phase variable more reliably than a Solidity developer manually inspecting the code. The same techniques could be used to identify further potential for reducing the storage requirements of smart contracts. If a variable required in the early phases of a dApp is not used in later phases, it could be recycled in later phases of the dApp to store different data, possibly unrelated to its original purpose. Program flow analysis could identify and automatically recycle such variables at compile time, effectively reducing the allocated storage.

8. Bibliography

- [1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. “Control-flow integrity”. In: *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*. 2005, pp. 340–353.
- [2] *Algorand – Website*. <https://www.algorand.foundation/>. Accessed 21-04-2024.
- [3] G. Almashaqbeh, F. Benhamouda, S. Han, D. Jaroslawicz, T. Malkin, A. Nicita, T. Rabin, A. Shah, and E. Tromer. “Gage MPC: Bypassing Residual Function Leakage for Non-Interactive MPC”. In: *Proc. Priv. Enhancing Technol.* 4 (2021), pp. 528–548.
- [4] *An In-Depth Look at the Parity Multisig Bug*. <https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/>. Accessed 31-01-2024.
- [5] *Analysis of the DAO exploit*. <https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>. Accessed 31-01-2024.
- [6] *Another Parity Wallet hack explained*. <https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c>. Accessed 31-01-2024.
- [7] *Arbitrum – Website*. <https://arbitrum.io/>. Accessed 30-04-2024.
- [8] D. W. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J. I. Pagter, N. P. Smart, and R. N. Wright. “From Keys to Databases - Real-World Applications of Secure Multi-Party Computation”. In: *Comput. J.* 12 (2018), pp. 1749–1771.
- [9] S. Arita and K. Tsurudome. “Construction of Threshold Public-Key Encryptions through Tag-Based Encryptions”. In: *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*. 2009, pp. 186–200.
- [10] *ARM Security Technology Building a Secure System using TrustZone Technology*. <https://developer.arm.com/documentation/PRD29-GENC-009492/>. Accessed 18-04-2024.

8. Bibliography

- [11] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. “More Efficient Oblivious Transfer Extensions with Security for Malicious Adversaries”. In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*. 2015, pp. 673–701.
- [12] G. Asharov and C. Orlandi. “Calling Out Cheaters: Covert Security with Public Verifiability”. In: *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*. 2012, pp. 681–698.
- [13] T. Attema, V. Dunning, M. H. Everts, and P. Langenkamp. “Efficient Compiler to Covert Security with Public Verifiability for Honest Majority MPC”. In: *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*. 2022, pp. 663–683.
- [14] Y. Aumann and Y. Lindell. “Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries”. In: *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007, Proceedings*. 2007, pp. 137–156.
- [15] *avalanche* – Website. <https://www.avax.network/>. Accessed 21-04-2024.
- [16] J. Baek and Y. Zheng. “Identity-Based Threshold Decryption”. In: *Public Key Cryptography - PKC 2004, 7th International Workshop on Theory and Practice in Public Key Cryptography, Singapore, March 1-4, 2004*. 2004, pp. 262–276.
- [17] R. Bahmani, M. Barbosa, F. Brassier, B. Portela, A. Sadeghi, G. Scerri, and B. Warinschi. “Secure Multiparty Computation from SGX”. In: *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*. 2017, pp. 477–497.
- [18] R. Bahmani, F. Brassier, G. Dessouky, P. Jauernig, M. Klimmek, A. Sadeghi, and E. Stapf. “CURE: A Security Architecture with Customizable and Resilient Enclaves”. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. 2021, pp. 1073–1090.
- [19] A. Banerjee, M. Clear, and H. Tewari. “zkHawk: Practical Private Smart Contracts from MPC-based Hawk”. In: *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services, BRAINS 2021, Paris, France, September 27-30, 2021*. 2021, pp. 245–248.
- [20] A. Banerjee and H. Tewari. “Multiverse of HawkNess: A Universally-Composable MPC-Based Hawk Variant”. In: *Cryptogr.* 3 (2022), p. 39.

8. Bibliography

- [21] C. Baum, J. H. Chiang, B. David, and T. K. Frederiksen. “Eagle: Efficient Privacy Preserving Smart Contracts”. In: *Financial Cryptography and Data Security - 27th International Conference, FC 2023, Bol, Brač, Croatia, May 1-5, 2023, Revised Selected Papers, Part I*. 2023, pp. 270–288.
- [22] C. Baum, I. Damgård, and C. Orlandi. “Publicly Auditable Secure Multi-Party Computation”. In: *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*. 2014, pp. 175–196.
- [23] D. Beaver. “Efficient Multiparty Protocols Using Circuit Randomization”. In: *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*. 1991, pp. 420–432.
- [24] D. Beaver, S. Micali, and P. Rogaway. “The Round Complexity of Secure Protocols (Extended Abstract)”. In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*. 1990, pp. 503–513.
- [25] A. Ben-Efraim, M. Nielsen, and E. Omri. “TurboSpeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing”. In: *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*. 2019, pp. 530–549.
- [26] M. Ben-Or, S. Goldwasser, and A. Wigderson. “Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)”. In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. 1988, pp. 1–10.
- [27] F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin. “Can a Public Blockchain Keep a Secret?” In: *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I*. 2020, pp. 260–290.
- [28] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A. Sadeghi. “The Guard’s Dilemma: Efficient Code-Reuse Attacks Against Intel SGX”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018, pp. 1213–1227.
- [29] *BnB Chain – Website*. <https://www.bnbchain.org/>. Accessed 21-04-2024.
- [30] *BNB Smart Chain*. <https://github.com/bnb-chain/whitepaper/blob/master/WHITEPAPER.md>. Accessed 18-01-2024.

8. Bibliography

- [31] P. Bogetoft et al. “Secure Multiparty Computation Goes Live”. In: *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*. 2009, pp. 325–343.
- [32] D. Boneh, X. Boyen, and S. Halevi. “Chosen Ciphertext Secure Public Key Threshold Encryption Without Random Oracles”. In: *Topics in Cryptology - CT-RSA 2006, The Cryptographers’ Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*. 2006, pp. 226–243.
- [33] D. Boneh and M. K. Franklin. “Identity-Based Encryption from the Weil Pairing”. In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*. 2001, pp. 213–229.
- [34] D. Boneh, D. B. Glass, D. Krashen, K. E. Lauter, S. Sharif, A. Silverberg, M. Tibouchi, and M. Zhandry. “Multiparty Non-Interactive Key Exchange and More From Isogenies on Elliptic Curves”. In: *J. Math. Cryptol.* 1 (2020), pp. 5–14.
- [35] X. Boyen and B. Waters. “Anonymous Hierarchical Identity-Based Encryption (Without Random Oracles)”. In: *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*. 2006, pp. 290–307.
- [36] E. Boyle, G. Couteau, N. Gilboa, and Y. Ishai. “Compressing Vector OLE”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018, pp. 896–912.
- [37] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl. “Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019, pp. 291–308.
- [38] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. “Efficient Pseudorandom Correlation Generators from Ring-LPN”. In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*. 2020, pp. 387–416.
- [39] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. “Efficient Pseudorandom Correlation Generators: Silent OT Extension and More”. In: *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III*. 2019, pp. 489–518.

8. Bibliography

- [40] E. Boyle, N. Gilboa, and Y. Ishai. “Breaking the Circuit Size Barrier for Secure Computation Under DDH”. In: *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*. 2016, pp. 509–539.
- [41] E. Boyle, N. Gilboa, Y. Ishai, H. Lin, and S. Tessaro. “Foundations of Homomorphic Secret Sharing”. In: *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*. 2018, 21:1–21:21.
- [42] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. “SANCTUARY: ARMing TrustZone with User-space Enclaves”. In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. 2019.
- [43] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017*. 2017.
- [44] J. Buchmann and H. C. Williams. “A Key-Exchange System Based on Imaginary Quadratic Fields”. In: *J. Cryptol.* 2 (1988), pp. 107–118.
- [45] J. V. Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018, pp. 991–1008.
- [46] S. S. Burra, E. Larraia, J. B. Nielsen, P. S. Nordholt, C. Orlandi, E. Orsini, P. Scholl, and N. P. Smart. “High-Performance Multi-party Computation for Binary Circuits Based on Oblivious Transfer”. In: *J. Cryptol.* 3 (2021), p. 34.
- [47] L. Caires and F. Pfenning. “Session Types as Intuitionistic Linear Propositions”. In: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. 2010, pp. 222–236.
- [48] L. Caires, F. Pfenning, and B. Toninho. “Linear logic propositions as session types”. In: *Math. Struct. Comput. Sci.* 3 (2016), pp. 367–423.
- [49] M. Campanelli, B. David, H. Khoshakhlagh, A. Konring, and J. B. Nielsen. “Encryption to the Future - A Paradigm for Sending Secret Messages to Future (Anonymous) Committees”. In: *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part III*. 2022, pp. 151–180.

8. Bibliography

- [50] C. Canella et al. “Fallout: Leaking Data on Meltdown-resistant CPUs”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019, pp. 769–784.
- [51] R. Canetti, B. Riva, and G. N. Rothblum. “Practical delegation of computation using multiple servers”. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. 2011, pp. 445–454.
- [52] *Cardano – Website*. <https://cardano.org/>. Accessed 21-04-2024.
- [53] J. Cartlidge, N. P. Smart, and Y. T. Alaoui. “MPC Joins The Dark Side”. In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. 2019, pp. 148–159.
- [54] I. Cascudo and J. S. Gundersen. “A Secret-Sharing Based MPC Protocol for Boolean Circuits with Good Amortized Complexity”. In: *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part II*. 2020, pp. 652–682.
- [55] D. Chaum, C. Crépeau, and I. Damgård. “Multiparty Unconditionally Secure Protocols (Extended Abstract)”. In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. 1988, pp. 11–19.
- [56] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song. “Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts”. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. 2019, pp. 185–200.
- [57] A. R. Choudhuri, M. Green, A. Jain, G. Kaptchuk, and I. Miers. “Fairness in an Unfair World: Fair Multiparty Computation from Public Bulletin Boards”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017, pp. 719–728.
- [58] M. J. Coblenz. “Obsidian: a safer blockchain programming language”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 2017, pp. 97–99.
- [59] M. J. Coblenz, G. Kambhatla, P. Koronkevich, J. L. Wise, C. Barnaby, J. Aldrich, J. Sunshine, and B. A. Myers. “User-Centered Programming Language Design in the Obsidian Smart Contract Language”. In: *CoRR* (2019). arXiv: 1912.04719.

8. Bibliography

- [60] M. J. Coblenz, R. Oei, T. Etzel, P. Koronkevich, M. Baker, Y. Bloem, B. A. Myers, J. Sunshine, and J. Aldrich. “Obsidian: Typestate and Assets for Safer Blockchain Programming”. In: *ACM Trans. Program. Lang. Syst.* 3 (2020), 14:1–14:82.
- [61] *CoinMarketCap – Charts*. <https://coinmarketcap.com/charts/>. Accessed 18-01-2024.
- [62] V. Costan and S. Devadas. “Intel SGX Explained”. In: *IACR Cryptol. ePrint Arch.* (2016), p. 86.
- [63] V. Costan, I. A. Lebedev, and S. Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016, pp. 857–874.
- [64] G. Couteau, P. Rindal, and S. Raghuraman. “Silver: Silent VOLE and Oblivious Transfer from Hardness of Decoding Structured LDPC Codes”. In: *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part III*. 2021, pp. 502–534.
- [65] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. “SPDZ_{2^k}: Efficient MPC mod 2^k for Dishonest Majority”. In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*. 2018, pp. 769–798.
- [66] K. Croman et al. “On Scaling Decentralized Blockchains - (A Position Paper)”. In: *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*. 2016, pp. 106–125.
- [67] I. Damgård, M. Geisler, and J. B. Nielsen. “From Passive to Covert Security at Low Cost”. In: *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*. 2010, pp. 128–145.
- [68] I. Damgård, M. Keller, E. Larraia, C. Miles, and N. P. Smart. “Implementing AES via an Actively/Covertly Secure Dishonest-Majority MPC Protocol”. In: *Security and Cryptography for Networks - 8th International Conference, SCN 2012, Amalfi, Italy, September 5-7, 2012. Proceedings*. 2012, pp. 241–263.
- [69] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. “Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits”. In: *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*. 2013, pp. 1–18.

8. Bibliography

- [70] I. Damgård, C. Orlandi, and M. Simkin. “Black-Box Transformations from Passive to Covert Security with Public Verifiability”. In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*. 2020, pp. 647–676.
- [71] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. 2012, pp. 643–662.
- [72] A. Das, S. Balzer, J. Hoffmann, F. Pfenning, and I. Santurkar. “Resource-Aware Session Types for Digital Contracts”. In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. 2021, pp. 1–16.
- [73] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A. Sadeghi. “FastKitten: Practical Smart Contracts on Bitcoin”. In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 2019, pp. 801–818.
- [74] R. DeLine and M. Fähndrich. “Typestates for Objects”. In: *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*. 2004, pp. 465–490.
- [75] W. Diffie and M. E. Hellman. “New directions in cryptography”. In: *IEEE Trans. Inf. Theory* 6 (1976), pp. 644–654.
- [76] S. Dittmer, Y. Ishai, S. Lu, and R. Ostrovsky. “Authenticated Garbling from Simple Correlations”. In: *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV*. 2022, pp. 57–87.
- [77] S. Dobson and S. D. Galbraith. “Trustless Groups of Unknown Order with Hyperelliptic Curves”. In: *IACR Cryptol. ePrint Arch.* (2020), p. 196.
- [78] S. Dziembowski, L. Eckey, and S. Faust. “FairSwap: How To Fairly Exchange Digital Goods”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018, pp. 967–984.
- [79] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková. “Multi-party Virtual State Channels”. In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*. 2019, pp. 625–656.

8. Bibliography

- [80] S. Dziembowski, S. Faust, and K. Hostáková. “General State Channel Networks”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2018, pp. 949–966.
- [81] *ECMAScript® 2023 Language Specification*. <https://262.ecma-international.org/>. Accessed 12-04-2024.
- [82] *Ethereum – The Merge*. <https://ethereum.org/en/roadmap/merge/>. Accessed 21-04-2024.
- [83] *Ethereum – Website*. <https://ethereum.org/>. Accessed 21-04-2024.
- [84] *Etherscan – Ethereum Average Gas Price Chart*. <https://etherscan.io/chart/gasprice>. 10 GWei per gas, 3152 USD per ETH. Accessed 03-05-2024.
- [85] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Financially Backed Covert Security”. In: *IACR Cryptol. ePrint Arch.* (2021), p. 1652.
- [86] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Financially Backed Covert Security”. In: *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part II*. 2022, pp. 99–129. **Part of this thesis.**
- [87] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Generic Compiler for Publicly Verifiable Covert Multi-Party Computation”. In: *IACR Cryptol. ePrint Arch.* (2021), p. 251.
- [88] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Generic Compiler for Publicly Verifiable Covert Multi-Party Computation”. In: *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*. 2021, pp. 782–811. **Part of this thesis.**
- [89] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Putting the Online Phase on a Diet: Covert Security from Short MACs”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 52.
- [90] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Putting the Online Phase on a Diet: Covert Security from Short MACs”. In: *Topics in Cryptology - CT-RSA 2023 - Cryptographers’ Track at the RSA Conference 2023, San Francisco, CA, USA, April 24-27, 2023, Proceedings*. 2023, pp. 360–386. **Part of this thesis.**
- [91] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Statement-Oblivious Threshold Witness Encryption”. In: *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*. 2023, pp. 17–32. **Part of this thesis.**
- [92] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Statement-Oblivious Threshold Witness Encryption”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 668.

8. Bibliography

- [93] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Threshold BBS+ From Pseudorandom Correlations”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1076.
- [94] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A. Sadeghi. “POSE: Practical Off-chain Smart Contract Execution”. In: *CoRR* (2022). arXiv: 2210.07110.
- [95] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A. Sadeghi. “POSE: Practical Off-chain Smart Contract Execution”. In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. 2023. **Part of this thesis.**
- [96] T. K. Frederiksen, M. Keller, E. Orsini, and P. Scholl. “A Unified Approach to MPC with Preprocessing Using OT”. In: *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*. 2015, pp. 711–735.
- [97] T. E. Gamal. “A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms”. In: *Advances in Cryptology, Proceedings of CRYPTO ’84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*. 1984, pp. 10–18.
- [98] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. “Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits”. In: *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*. 2013, pp. 40–49.
- [99] S. Garg, C. Gentry, A. Sahai, and B. Waters. “Witness encryption and its applications”. In: *Symposium on Theory of Computing Conference, STOC’13, Palo Alto, CA, USA, June 1-4, 2013*. 2013, pp. 467–476.
- [100] C. Gentry. “Practical Identity-Based Encryption Without Random Oracles”. In: *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*. 2006, pp. 445–464.
- [101] C. Gentry, S. Halevi, B. Magri, J. B. Nielsen, and S. Yakoubov. “Random-Index PIR and Applications”. In: *Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021, Proceedings, Part III*. 2021, pp. 32–61.
- [102] C. Gentry, A. B. Lewko, and B. Waters. “Witness Encryption from Instance Independent Assumptions”. In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*. 2014, pp. 426–443.

8. Bibliography

- [103] M. George and S. Kamara. “Adversarial Level Agreements for Two-Party Protocols”. In: *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*. 2022, pp. 816–830.
- [104] O. Goldreich, S. Micali, and A. Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. 1987, pp. 218–229.
- [105] S. Goldwasser, Y. T. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. “How to Run Turing Machines on Encrypted Data”. In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*. 2013, pp. 536–553.
- [106] V. Goyal, A. Kothapalli, E. Masserova, B. Parno, and Y. Song. “Storing and Retrieving Secrets on a Blockchain”. In: *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part I*. 2022, pp. 252–282.
- [107] V. Goyal, E. Masserova, B. Parno, and Y. Song. “Blockchains Enable Non-interactive MPC”. In: *Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021, Proceedings, Part II*. 2021, pp. 162–193.
- [108] V. Goyal, P. Mohassel, and A. D. Smith. “Efficient Two Party and Multi Party Computation Against Covert Adversaries”. In: *Advances in Cryptology - EURO-CRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*. 2008, pp. 289–306.
- [109] D. Gupta, B. Mood, J. Feigenbaum, K. R. B. Butler, and P. Traynor. “Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation”. In: *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*. 2016, pp. 302–318.
- [110] C. Hazay and Y. Lindell. “Efficient Oblivious Polynomial Evaluation with Simulation-Based Security”. In: *IACR Cryptol. ePrint Arch.* (2009), p. 459.
- [111] C. Hazay and Y. Lindell. “Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries”. In: *Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008*. 2008, pp. 155–175.

8. Bibliography

- [112] C. Hazay, P. Scholl, and E. Soria-Vazquez. “Low Cost Constant Round MPC Combining BMR and Oblivious Transfer”. In: *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*. 2017, pp. 598–628.
- [113] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo. “Using innovative instructions to create trustworthy software solutions”. In: *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. 2013, p. 11.
- [114] K. Honda. “Types for Dyadic Interaction”. In: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*. 1993, pp. 509–523.
- [115] K. Honda, V. T. Vasconcelos, and M. Kubo. “Language Primitives and Type Discipline for Structured Communication-Based Programming”. In: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*. 1998, pp. 122–138.
- [116] C. Hong, J. Katz, V. Kolesnikov, W. Lu, and X. Wang. “Covert Security with Public Verifiability: Faster, Leaner, and Simpler”. In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*. 2019, pp. 97–121.
- [117] J. van den Hooff, M. F. Kaashoek, and N. Zeldovich. “VerSum: Verifiable Computations over Large Public Logs”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. 2014, pp. 1304–1316.
- [118] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. “Extending Oblivious Transfers Efficiently”. In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. 2003, pp. 145–161.
- [119] Y. Ishai, M. Prabhakaran, and A. Sahai. “Founding Cryptography on Oblivious Transfer - Efficiently”. In: *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008, Proceedings*. 2008, pp. 572–591.
- [120] H. A. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten. “Arbitrum: Scalable, private smart contracts”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018, pp. 1353–1370.

8. Bibliography

- [121] S. Kanjalkar, Y. Zhang, S. Gandlur, and A. Miller. “Publicly Auditable MPC-as-a-Service with succinct verification and universal setup”. In: *IEEE European Symposium on Security and Privacy Workshops, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. 2021, pp. 386–411.
- [122] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang. “Optimizing Authenticated Garbling for Faster Secure Two-Party Computation”. In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*. 2018, pp. 365–391.
- [123] M. Keller, E. Orsini, and P. Scholl. “MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, pp. 830–842.
- [124] J. C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* 7 (1976), pp. 385–394.
- [125] B. Knott, S. Venkataraman, A. Y. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten. “CrypTen: Secure Multi-Party Computation Meets Machine Learning”. In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 2021, pp. 4961–4973.
- [126] V. Kolesnikov and A. J. Malozemoff. “Public Verifiability in the Covert Model (Almost) for Free”. In: *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II*. 2015, pp. 210–235.
- [127] V. Kolesnikov, P. Mohassel, B. Riva, and M. Rosulek. “Richer Efficiency/Security Trade-offs in 2PC”. In: *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part I*. 2015, pp. 229–259.
- [128] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. “Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts”. In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 2016, pp. 839–858.
- [129] K. A. Küçük, A. Paverd, A. C. Martin, N. Asokan, A. Simpson, and R. Ankele. “Exploring the use of Intel SGX for Secure Many-Party Applications”. In: *Proceedings of the 1st Workshop on System Software for Trusted Execution, Sys-TEX@Middleware 2016, Trento, Italy, December 12, 2016*. 2016, 5:1–5:6.

8. Bibliography

- [130] A. Küpçü and P. Mohassel. “Fast Optimistically Fair Cut-and-Choose 2PC”. In: *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers*. 2016, pp. 208–228.
- [131] E. Larraia, E. Orsini, and N. P. Smart. “Dishonest Majority Multi-Party Computation for Binary Circuits”. In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*. 2014, pp. 495–512.
- [132] Y. Lindell. “Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries”. In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*. 2013, pp. 1–17.
- [133] Y. Lindell, E. Oxman, and B. Pinkas. “The IPS Compiler: Optimizations, Variants and Concrete Efficiency”. In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. 2011, pp. 259–276.
- [134] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. “ARMageddon: Cache Attacks on Mobile Devices”. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016, pp. 549–564.
- [135] J. Liu, T. Jager, S. A. Kakvi, and B. Warinschi. “How to build time-lock encryption”. In: *Des. Codes Cryptogr.* 11 (2018), pp. 2549–2586.
- [136] Y. Liu, J. Lai, Q. Wang, X. Qin, A. Yang, and J. Weng. “Robust Publicly Verifiable Covert Security: Limited Information Leakage and Guaranteed Correctness with Low Overhead”. In: *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part I*. 2023, pp. 272–301.
- [137] Y. Liu, Q. Wang, and S. Yiu. “Making Private Function Evaluation Safer, Faster, and Simpler”. In: *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part I*. 2022, pp. 349–378.
- [138] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller. “HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and its Application to Anonymous Communication”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019, pp. 887–903.

8. Bibliography

- [139] P. D. MacKenzie, M. K. Reiter, and K. Yang. “Alternatives to Non-malleability: Definitions, Constructions, and Applications (Extended Abstract)”. In: *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*. 2004, pp. 171–190.
- [140] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. “Innovative instructions and software model for isolated execution”. In: *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. 2013, p. 10.
- [141] R. C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. 1987, pp. 369–378.
- [142] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry. “Sprites and State Channels: Payment Networks that Go Faster Than Lightning”. In: *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. 2019, pp. 508–526.
- [143] P. Mohassel and B. Riva. “Garbled Circuits Checking Garbled Circuits: More Efficient and Secure Two-Party Computation”. In: *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*. 2013, pp. 36–53.
- [144] MPC Alliance. <https://www.mpcalliance.org/>. Accessed 01-02-2024.
- [145] S. Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: (2008).
- [146] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. “A New Approach to Practical Active-Secure Two-Party Computation”. In: *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. 2012, pp. 681–700.
- [147] J. B. Nielsen, T. Schneider, and R. Trifiletti. “Constant Round Maliciously Secure 2PC with Function-independent Preprocessing using LEGO”. In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. 2017.
- [148] Optimism – Website. <https://www.optimism.io/>. Accessed 30-04-2024.
- [149] A. J. Paverd, A. P. Martin, and I. Brown. “Privacy-enhanced bi-directional communication in the Smart Grid using trusted computing”. In: *2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3-6, 2014*. 2014, pp. 872–877.

8. Bibliography

- [150] K. Pietrzak. “Simple Verifiable Delay Functions”. In: *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*. 2019, 60:1–60:15.
- [151] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. “Secure Two-Party Computation Is Practical”. In: *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*. 2009, pp. 250–267.
- [152] D. Richter, D. Kretzler, P. Weisenburger, G. Salvaneschi, S. Faust, and M. Mezini. “Prisma : A Tierless Language for Enforcing Contract-client Protocols in Decentralized Applications”. In: *ACM Trans. Program. Lang. Syst.* 3 (2023), 17:1–17:41. **Part of this thesis**.
- [153] D. Richter, D. Kretzler, P. Weisenburger, G. Salvaneschi, S. Faust, and M. Mezini. “Prisma: A Tierless Language for Enforcing Contract-Client Protocols in Decentralized Applications (Extended Version)”. In: *CoRR* (2022). arXiv: 2205.07780.
- [154] R. L. Rivest, A. Shamir, and D. A. Wagner. “Time-lock puzzles and timed-release crypto”. In: (1996).
- [155] P. Scholl, M. Simkin, and L. Siniscalchi. “Multiparty Computation with Covert Security and Public Verifiability”. In: *3rd Conference on Information-Theoretic Cryptography, ITC 2022, July 5-7, 2022, Cambridge, MA, USA*. 2022, 8:1–8:13.
- [156] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova. “Distributed Vector-OLE: Improved Constructions and Implementation”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019, pp. 1055–1072.
- [157] A. Shamir. “How to Share a Secret”. In: *Commun. ACM* 11 (1979), pp. 612–613.
- [158] A. Shamir. “Identity-Based Cryptosystems and Signature Schemes”. In: *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*. 1984, pp. 47–53.
- [159] M. Shih, S. Lee, T. Kim, and M. Peinado. “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs”. In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. 2017.
- [160] *Solana – Website*. <https://solana.com/>. Accessed 21-04-2024.
- [161] *Solidity Documentation*. <https://docs.soliditylang.org/en/v0.8.24/>. Accessed 31-01-2024.
- [162] *Starknet – Website*. <https://www.starknet.io/>. Accessed 30-04-2024.

8. Bibliography

- [163] S. Steffen, B. Bichsel, R. Baumgartner, and M. T. Vechev. “ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs”. In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. 2022, pp. 179–197.
- [164] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. T. Vechev. “zkay: Specifying and Enforcing Data Privacy in Smart Contracts”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019, pp. 1759–1776.
- [165] R. E. Strom and S. Yemini. “Typestate: A Programming Language Concept for Enhancing Software Reliability”. In: *IEEE Trans. Software Eng.* 1 (1986), pp. 157–171.
- [166] *Tron – Website*. <https://tron.network/>. Accessed 21-04-2024.
- [167] *TRON Whitepaper: Advanced Decentralized Blockchain Platform*. https://tron.network/static/doc/white_paper_v_2_0.pdf. Accessed 18-01-2024.
- [168] *VISA Fact Sheet*. <https://www.visa.co.uk/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf>. Accessed 17-01-2024.
- [169] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. “Conclave: secure multi-party computation on big data”. In: *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*. 2019, 3:1–3:18.
- [170] P. Wadler. “Propositions as sessions”. In: *ACM SIGPLAN International Conference on Functional Programming, ICFP’12, Copenhagen, Denmark, September 9-15, 2012*. 2012, pp. 273–286.
- [171] X. Wang, S. Ranellucci, and J. Katz. “Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2017, pp. 21–37.
- [172] B. Wesolowski. “Efficient Verifiable Delay Functions”. In: *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*. 2019, pp. 379–407.
- [173] G. Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper (Paris Version)* (2024).
- [174] K. Wüst, L. Diana, K. Kostianen, G. Karame, S. Matetic, and S. Capkun. “Bitcontracts: Supporting Smart Contracts in Legacy Blockchains”. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. 2021.

8. Bibliography

- [175] K. Wüst, S. Matetic, S. Egli, K. Kostianen, and S. Capkun. “ACE: Asynchronous and Concurrent Execution of Complex Smart Contracts”. In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. 2020, pp. 587–600.
- [176] B. Xu, D. Luthra, Z. Cole, and N. Blakely. “EOS: An Architectural, Performance, and Economic Analysis”. In: (2018).
- [177] A. Yakovenko. “Solana: A new architecture for a high performance blockchain”. In: (2018).
- [178] K. Yang, X. Wang, and J. Zhang. “More Efficient MPC from Improved Triple Generation and Authenticated Garbling”. In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. 2020, pp. 1627–1646.
- [179] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang. “Ferret: Fast Extension for Correlated OT with Small Communication”. In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. 2020, pp. 1607–1626.
- [180] A. C. Yao. “How to Generate and Exchange Secrets (Extended Abstract)”. In: *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. 1986, pp. 162–167.
- [181] *Zengo Wallet*. <https://zengo.com/>. Accessed 01-02-2024.
- [182] R. Zhu, C. Ding, and Y. Huang. “Efficient Publicly Verifiable 2PC over a Blockchain with Applications to Financially-Secure Computations”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 2019, pp. 633–650.
- [183] *zkSync – Website*. <https://zksync.io/>. Accessed 30-04-2024.
- [184] G. Zyskind, O. Nathan, and A. Pentland. “Enigma: Decentralized Computation Platform with Guaranteed Privacy”. In: *CoRR* (2015). arXiv: 1506.03471.

List of Figures

3.1. Exemplary bisection search over two lists deviating in the last element.	38
4.1. Different ways to instantiate statement-oblivious threshold witness encryption.	50
5.1. Overview of the POSE system including the processes of contract creation (blue) and the contract execution (green).	56
6.1. Solidity code of a tic-tac-toe contract with a funding phase and a payout phase (left). The code implicitly defines a finite state machine (right). Red dashed arrows illustrate unintended transitions, e.g., premature calls of the <code>Payout</code> function. Line 14 of the contract (underlined in red) prevents such unintended payouts.	62
6.2. Prisma dApp implementing the tic-tac-toe case study.	64
6.3. Simplified Obsidian smart contract of the tic-tac-toe case study. . .	67
6.4. Simplified Nomos smart contract of the tic-tac-toe case study. . . .	67

List of Abbreviations

A-TIBE	Anonymous Threshold Identity-based Encryption
CCA	Chosen Ciphertext Attacks
DAO	Decentralized Autonomous Organization
DBDH	Decision bilinear Diffie-Hellman
EVM	Ethereum Virtual Machine
eWEB	Extractable Witness Encryption on Blockchains
FBC	Financially Backed Verifiable Covert
FSM	Finite State Machine
HSS	Homomorphic Secret Sharing
IBE	Identity-based Encryption
IECF	Intermediate Explicit Cheat Formulation
IND-CCA	Indistinguishably Under Chosen Ciphertext Attacks
IND-CPA	Indistinguishably Under Chosen Plaintext Attacks
MAC	Message Authentication Code
MPC	Multiparty Computation
NP	Nondeterministic Polynomial-Time
OT	Oblivious Transfer
O-TTBE	Oblivious Threshold Tag-Based Encryption
PoM	Proof of Misbehavior
PoS	Proof-of-Stake
PoW	Proof-of-Work
PPT	Probabilistic Polynomial-Time
PVC	Publicly Verifiable Covert
SECF	Strong Explicit Cheat Formulation
SO-TWE	Statement Oblivious Threshold Witness Encryption
TBE	Tag-based Encryption
TEE	Trusted Execution Environment
TLP	Time-Lock Puzzle
TTBE	Threshold Tag-based Encryption
TTP	Trusted Third Party
USD	United States Dollar
V-TTP	Virtual Trusted Third Party

A. Generic Compiler for Publicly Verifiable Covert Multi-Party Computation

This chapter corresponds to the following publication. The full version is available at [87].

- [88] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Generic Compiler for Publicly Verifiable Covert Multi-Party Computation”. In: *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*. 2021, pp. 782–811. **Part of this thesis.**

Generic Compiler for Publicly Verifiable Covert Multi-Party Computation

Sebastian Faust¹, Carmit Hazay², David Kretzler¹, and Benjamin Schlosser¹

¹ Technical University of Darmstadt, Germany

`{first.last}@tu-darmstadt.de`

² Bar-Ilan University, Israel

`carmit.hazay@biu.ac.il`

Abstract. Covert security has been introduced as a compromise between semi-honest and malicious security. In a nutshell, covert security guarantees that malicious behavior can be detected by the honest parties with some probability, but in case detection fails all bets are off. While the security guarantee offered by covert security is weaker than full-fledged malicious security, it comes with significantly improved efficiency. An important extension of covert security introduced by Asharov and Orlandi (ASIACRYPT'12) is *public verifiability*, which allows the honest parties to create a publicly verifiable certificate of malicious behavior. Public verifiability significantly strengthens covert security as the certificate allows punishment via an external party, e.g., a judge.

Most previous work on publicly verifiable covert (PVC) security focuses on the two-party case, and the multi-party case has mostly been neglected. In this work, we introduce a novel compiler for multi-party PVC secure protocols with no private inputs. The class of supported protocols includes the preprocessing of common multi-party computation protocols that are designed in the offline-online model. Our compiler leverages time-lock encryption to offer high probability of cheating detection (often also called deterrence factor) independent of the number of involved parties. Moreover, in contrast to the only earlier work that studies PVC in the multi-party setting (CRYPTO'20), we provide the first full formal security analysis.

Keywords: Covert Security · Multi-Party Computation · Public Verifiability · Time-Lock Puzzles

1 Introduction

Secure multi-party computation (MPC) allows a set of n parties P_i to jointly compute a function f on their inputs such that nothing beyond the output of that function is revealed. Privacy of the inputs and correctness of the outputs need to be guaranteed even if some subset of the parties is corrupted by an adversary. The two most prominent adversarial models considered in the literature are the *semi-honest* and *malicious* adversary model. In the semi-honest model, the adversary is passive and the corrupted parties follow the protocol

description. Hence, the adversary only learns the inputs and incoming/outgoing messages including the internal randomness of the corrupted parties. In contrast, the adversarial controlled parties can arbitrarily deviate from the protocol specification under malicious corruption.

Since in most cases it seems hard (if not impossible) to guarantee that a corrupted party follows the protocol description, malicious security is typically the desired security goal for the design of multi-party computation protocols. Unfortunately, compared to protocols that only guarantee semi-honest security, protection against malicious adversaries results into high overheads in terms of communication and computation complexity. For protocols based on distributed garbling techniques in the oblivious transfer (OT)-hybrid model, the communication complexity is inflated by a factor of $\frac{s}{\log |\mathcal{C}|}$ [WRK17b], where \mathcal{C} is the computed circuit and s is a statistical security parameter. For secret sharing-based protocols, Hazay et al. [HVW20] have recently shown a constant communication overhead over the semi-honest GMW-protocol [GMW87]. In most techniques, the computational overhead grows with an order of s .

In order to mitigate the drawbacks of the overhead required for malicious secure function evaluation, one approach is to split protocols into an input-independent offline and an input-dependent online phase. The input-independent offline protocol carries out pre-computations that are utilized to speed up the input-dependent online protocol which securely evaluates the desired function. Examples for such offline protocols are the circuit generation of garbling schemes as in authenticated garbling [WRK17a, WRK17b] or the generation of correlated randomness in form of Beaver triples [Bea92] in secret sharing-based protocols such as in SPDZ [DPSZ12]. The main idea of this approach is that the offline protocol can be executed continuously *in the background* and the online protocol is executed ad-hoc once input data becomes available or output data is required. Since the performance requirements for the online protocol are usually much stricter, the offline part should cover the most expensive protocol steps, as for example done in [WRK17a, WRK17b] and [DPSZ12].

A middle ground between the design goals of security and efficiency has been proposed with the notion of *covert security*. Introduced by Aumann and Lindell [AL07], covert security allows the adversary to take full control over a party and let her deviate from the protocol specification in an arbitrary way. The protocol, however, is designed in such a way that honest parties can detect cheating with some probability ϵ (often called the deterrence factor). However, if cheating is not detected all bets are off. This weaker security notion comes with the benefit of significantly improved efficiency, when compared to protocols in the full-fledged malicious security model. The motivation behind covert security is that in many real-world scenarios, parties are able to actively deviate from the protocol instructions (and as such are not semi-honest), but due to reputation concerns only do so if they are not caught. In the initial work of Aumann and Lindell, the focus was on the two-party case. This has been first extended to the multi-party case by Goyal et al. [GMS08] and later been adapted to a different line of MPC protocols by Damgård et al. [DKL⁺13].

While the notion of covert security seems appealing at first glance it has one important shortcoming. If an honest party detects cheating, then she cannot reliably transfer her knowledge to other parties, which makes the notion of covert security significantly less attractive for many applications. This shortcoming of covert security was first observed by Asharov and Orlandi [AO12], and addressed with the notion of *public verifiability*. Informally speaking, public verifiability guarantees that if an honest party detects cheating, she can create a certificate that uniquely identifies the cheater, and can be verified by an external party. Said certificate can be used to punish cheaters for misbehavior, e.g., via a smart contract [ZDH19], thereby disincentivizing misbehavior.

Despite being a natural security notion, there has been relatively little work on covert security with public verifiability. In particular, starting with the work of Asharov and Orlandi [AO12] most works have explored publicly verifiable covert security in the two-party setting [KM15, HKK⁺19, ZDH19, DOS20]. These works use a publicly checkable cut-and-choose approach for secure two-party computation based on garbled circuits. Here a random subset of size $t-1$ out of t garbled circuits is opened to verify if cheating occurred, while the remaining unopened garbled circuit is used for the actual secure function evaluation. The adversary needs to guess which circuit is used for the final evaluation and only cheat in this particular instance. If her guess is false, she will be detected. Hence, there is a deterrence factor of $\frac{t-1}{t}$.

For the extension to the multi-party case of covert security even less is known. Prior work mainly focuses on the restricted version of covert security that does not offer public verifiability [GMS08, DGN10, LOP11, DKL⁺13]. The only work that we are aware of that adds public verifiability to covert secure multi-party computation protocols is the recent work of Damgård et al. [DOS20]. While [DOS20] mainly focuses on a compiler for the two-party case, they also sketch how their construction can be extended to the multi-party setting.

1.1 Our Contribution

In contrast to most prior research, we focus on the multi-party setting. Our main contribution is a novel compiler for transforming input-independent multi-party computation protocols with semi-honest security into protocols that offer covert security with public verifiability. Our construction achieves a high deterrence factor of $\frac{t-1}{t}$, where t is the number of semi-honest instances executed in the cut-and-choose protocol. In contrast, the only prior work that sketches a solution for publicly verifiable covert security for the multi-part setting [DOS20] achieves $\approx \frac{t-1}{nt}$, which in particular for a large number of parties n results in a low deterrence factor. [DOS20] states that the deterrence factor can be increased at the cost of multiple protocol repetitions, which results into higher complexity and can be abused to amplify denial-of-service attacks. A detail discussion of the main differences between [DOS20] and our work is given in Section 6. We emphasize that our work is also the first that provides a full formal security proof of the multi-party case in the model of covert security with public verifiability.

Our results apply to a large class of input-independent offline protocols for carrying out pre-computation. Damgård et al. [DOS20] have shown that an offline-online protocol with a publicly verifiable covert secure offline phase and a maliciously secure online phase constitutes a publicly verifiable covert secure protocol in total. Hence, by applying our compiler to a passively secure offline protocol and combining it with an actively secure online protocol, we obtain a publicly verifiable covert secure protocol in total. Since offline protocols are often the most expensive part of the secure multi-party computation protocol, e.g., in protocols like [YWZ20] and [DPSZ12], our approach has the potential of significantly improving efficiency of multi-party computation protocols in terms of computation and communication overhead.

An additional contribution of our work (which is of independent interest) is to introduce a novel mechanism for achieving public verifiability in protocols with covert security. Our approach is based on *time-lock encryption* [RSW96, MT19, MMV11, BGJ⁺16], a primitive that enables encryption of messages into the future and has previously been discussed in the context of delayed digital cash payments, sealed-bid auctions, key escrow, and e-voting. Time-lock encryption can be used as a building block to guarantee that in case of malicious behavior each honest party can construct a publicly verifiable cheating certificate without further interaction. The use of time-lock puzzles in a simulation-based security proof requires us to overcome several technical challenges that do not occur for proving game-based security notions.

In order to achieve efficient verification of the cheating certificates, we also show how to add verifiability to the notion of time-lock encryption by using techniques from verifiable delay functions [BBBF18]. While our construction can be instantiated with any time-lock encryption satisfying our requirements, we present a concrete extension of the RSW time-lock encryption scheme. Since RSW-based time-lock encryption [RSW96, MT19] requires a one-time trusted setup, an instantiation of our construction using the RSW-based time-lock encryption inherits this assumption. We can implement the one-time trusted setup using a maliciously secure multi-party computation protocol similar to the MPC ceremony used, e.g., by the cryptocurrency ZCash.

1.2 Technical Overview

In this section, we give a high-level overview of the main techniques used in our work. To this end, we start by briefly recalling how covert security is typically achieved. Most covert secure protocols take a semi-honest protocol and execute t instances of it in parallel. They then check the correctness of $t - 1$ randomly chosen instances by essentially revealing the used inputs and randomness and finally take the result of the last unopened execution as protocol output. The above requires that (a) checking the correctness of the $t - 1$ instances can be carried out efficiently, and (b) the private inputs of the parties are not revealed.

In order to achieve the first goal, one common approach is to derandomize the protocol, i.e., let the parties generate a random seed from which they derive their internal randomness. Once the protocol is derandomized, correctness can

efficiently be checked by the other parties. To achieve the second goal, the protocol is divided into an offline and an online protocol as described above. The output of the offline phase (e.g., a garbling scheme) is just some correlated randomness. As this protocol is input-independent, the offline phase does not leak information about the parties' private inputs. The online phase (e.g., evaluating a garbled circuit) is maliciously secure and hence protects the private inputs.

Public verifiability. To add public verifiability to the above-described approach, the basic idea is to let the parties sign all transcripts that have been produced during the protocol execution. This makes them accountable for cheating in one of the semi-honest executions. One particular challenge for public verifiability is to ensure that once a malicious party notices that its cheating attempt will be detected it cannot prevent (e.g., by aborting) the creation of a certificate proving its misbehavior. Hence, the trivial idea of running a shared coin tossing protocol to select which of the instances will be checked does not work because the adversary can abort before revealing her randomness and inputs used in the checked instances. To circumvent this problem, the recent work of Damgård et al. [DOS20] proposes the following technique. Each party locally chooses a subset I of the t semi-honest instances whose computation it wants to check (this is often called a watchlist [IPS08]). Next, it obviously asks the parties to explain their execution in those instances (i.e., by revealing the random coins used in the protocol execution). While this approach works well in the two-party case, in the multi-party case it either results in a low deterrence factor or requires that the protocol execution is repeated many times. This is due to the fact that each party chooses its watchlist independently; in the worst case, all watchlists are mutually disjoint. Hence, the size of each watchlist is set to be lower or equal than $\frac{t-1}{n}$ (resulting in a deterrence factor of $\frac{t-1}{nt}$) to guarantee that one instance remains unchecked or parties repeat the protocol several times until there is a protocol execution with an unchecked instance.

Public verifiability from time-lock encryption. Our approach avoids the above shortcomings by using time-lock encryption. Concretely, we follow the shared coin-tossing approach mentioned above but prevent the rushing attack by locking the shared coin (selecting which semi-honest executions shall be opened) and the seeds of the opened executions in time-lock encryption. Since the time-lock ciphertexts are produced before the selection-coin is made public, it will be too late for the adversary to abort the computation. Moreover, since the time-lock encryption can be solved even without the participation of the adversary, the honest parties can produce a publicly verifiable certificate to prove misbehavior. This approach has the advantage that we can always check all but one instance of the semi-honest executions, thereby significantly improving the deterrence factor and the overall complexity. One may object that solving time-lock encryption adds additional computational overhead to the honest parties. We emphasize, however, that the time-lock encryption has to be solved only in the pessimistic case when one party aborts after the puzzle generation. Moreover, in our construction, the time-lock parameter can be chosen rather small, since

the encryption has to hide the selection-coin and the seeds only for two communication rounds. See section 6 for a more detailed analysis of the overhead introduced by the time-lock puzzle generation and a comparison to prior work.

Creating the time-lock encryption. There are multiple technical challenges that we need to address to make the above idea work. First, current constructions of time-lock encryption matching our requirements require a trusted setup for generating the public parameters. In particular, we need to generate a strong RSA modulus N without leaking its factorization, and produce a base-puzzle that later can be used for efficiency reasons. Both of these need to be generated just once and can be re-used for all protocol executions. Hence, one option is to replace the trusted setup by a maliciously secure MPC similar to what has been done for the MPC ceremony used by the cryptocurrency ZCash. Another alternative is to investigate if time-lock puzzles matching the requirements of our compiler can be constructed from hidden order groups with public setup such as ideal class groups of imaginary quadratic fields [BW88] or Jacobians of hyperelliptic curves [DG20]. An additional challenge is that we cannot simply time-lock the seeds of all semi-honest protocol executions (as one instance needs to remain unopened). To address this problem, we use a maliciously secure MPC protocol to carry out the shared coin-tossing protocol and produce the time-lock encryptions of the seeds for the semi-honest protocol instance that are later opened. We emphasize that the complexity of this step only depends on t and n , and is in particular independent of the complexity of the functionality that we want to compute. Hence, for complex functionalities the costs of the maliciously secure puzzle generation are amortized over the protocol costs ³.

2 Secure Multi-Party Computation

Secure computation in the standalone model is defined via the real world/ideal world paradigm. In the real world, all parties interact in order to jointly execute the protocol Π . In the ideal world, the parties send their inputs to a trusted party called ideal functionality and denoted by \mathcal{F} which computes the desired function f and returns the result back to the parties. It is easy to see that in the ideal world the computation is correct and reveals only the intended information by definition. The security of a protocol Π is analyzed by comparing the ideal-world execution with the real-world execution. Informally, protocol Π is said to securely realize \mathcal{F} if for every real-world adversary \mathcal{A} , there exists an ideal-world adversary \mathcal{S} such that the joint output distribution of the honest parties and the adversary \mathcal{A} in the real-world execution of Π is indistinguishable from the joint output distribution of the honest parties and \mathcal{S} in the ideal-world execution.

We denote the number of parties executing a protocol Π by n . Let $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$, where $f = (f_1, \dots, f_n)$, be the function realized by Π .

³ Concretely, for each instantiation we require two exponentiations and a small number of symmetric key encryptions. The latter can be realized using tailored MPC-ciphers like LowMC [ARS⁺15].

For every input vector $\bar{x} = (x_1, \dots, x_n)$ the output vector is $\bar{y} = (f_1(\bar{x}), \dots, f_n(\bar{x}))$ and the i -th party P_i with input x_i obtains output $f_i(\bar{x})$.

An adversary can corrupt any subset $I \subseteq [n]$ of parties. We further set $\text{REAL}_{\Pi, \mathcal{A}(z), I}(\bar{x}, 1^\kappa)$ to be the output vector of the protocol execution of Π on input $\bar{x} = (x_1, \dots, x_n)$ and security parameter κ , where the adversary \mathcal{A} on auxiliary input z corrupts the parties $I \subseteq [n]$. By $\text{OUTPUT}_i(\text{REAL}_{\Pi, \mathcal{A}(z), I}(\bar{x}, 1^\kappa))$, we specify the output of party P_i for $i \in [n]$.

2.1 Covert Security

Aumann and Lindell introduced the notion of *covert security with ϵ -deterrence factor* in 2007 [AL07]. We focus on the strongest given formulation of covert security that is the *strong explicit cheat formulation*, where the ideal-world adversary only learns the honest parties' inputs if cheating is undetected. However, we slightly modify the original notion of covert security to capture realistic effects that occur especially in input-independent protocols and are disregarded by the notion of [AL07]. The changes are explained and motivated below.

As in the standard secure computation model, the execution of a real-world protocol is compared to the execution within an ideal world. The real world is exactly the same as in the standard model but the ideal model is slightly adapted in order to allow the adversary to cheat. Cheating will be detected by some fixed probability ϵ , which is called the deterrence factor. Let $\epsilon : \mathbb{N} \rightarrow [0, 1]$ be a function, then the execution in the ideal model works as follows.

Inputs: Each party obtains an input; the i^{th} party's input is denoted by x_i . We assume that all inputs are of the same length. The adversary receives an auxiliary input z .

Send inputs to trusted party: Any honest party P_j sends its received input x_j to the trusted party. The corrupted parties, controlled by \mathcal{S} , may either send their received input, or send some other input of the same length to the trusted party. This decision is made by \mathcal{S} and may depend on the values x_i for $i \in I$ and auxiliary input z . If there are no inputs, the parties send ok_i instead of their inputs to the trusted party.

Trusted party answers adversary: If the trusted party receives inputs from all parties, the trusted party computes $(y_1, \dots, y_m) = f(\bar{w})$ and sends y_i to \mathcal{S} for all $i \in I$.

Abort options: If the adversary sends **abort** to the trusted party as additional input (before or after the trusted party sends the potential output to the adversary), then the trusted party sends **abort** to all the honest parties and halts. If a corrupted party sends additional input $w_i = \text{corrupted}_i$ to the trusted party, then the trusted party sends corrupted_i to all of the honest parties and halts. If multiple parties send corrupted_i , then the trusted party disregards all but one of them (say, the one with the smallest index i). If both corrupted_i and **abort** messages are sent, then the trusted party ignores the corrupted_i message.

Attempted cheat option: If a corrupted party sends additional input $w_i = \text{cheat}_i$ to the trusted party (as above: if there are several messages $w_i = \text{cheat}_i$

ignore all but one - say, the one with the smallest index i), then the trusted party works as follows:

1. With probability ϵ , the trusted party sends `corruptedi` to the adversary and all of the honest parties.
2. With probability $1 - \epsilon$, the trusted party sends `undetected` to the adversary along with the honest parties inputs $\{x_j\}_{j \notin I}$. Following this, the adversary sends the trusted party `abort` or output values $\{y_j\}_{j \notin I}$ of its choice for the honest parties. If the adversary sends `abort`, the trusted party sends `abort` to all honest parties. Otherwise, for every $j \notin I$, the trusted party sends y_j to P_j .

The ideal execution then ends at this point. Otherwise, if no w_i equals `aborti`, `corruptedi` or `cheati`, the ideal execution continues below.

Trusted party answers honest parties: If the trusted party did not receive `corruptedi`, `cheati` or `abort` from the adversary or a corrupted party then it sends y_j for all honest parties P_j (where $j \notin I$).

Outputs: An honest party always outputs the message it obtained from the trusted party. The corrupted parties outputs nothing. The adversary \mathcal{S} outputs any arbitrary (probabilistic) polynomial-time computable function of the initial inputs $\{x_i\}_{i \in I}$, the auxiliary input z , and the received messages.

We denote by $\text{IDEALC}_{f, \mathcal{S}(z), I}^\epsilon(\bar{x}, 1^\kappa)$ the output of the honest parties and the adversary in the execution of the ideal model as defined above, where \bar{x} is the input vector and the adversary \mathcal{S} runs on auxiliary input z .

Definition 1 (Covert security with ϵ -deterrent). *Let f, Π , and ϵ be as above. Protocol Π is said to securely compute f in the presence of covert adversaries with ϵ -deterrent if for every non-uniform probabilistic polynomial-time adversary \mathcal{A} for the real model, there exists a non-uniform probabilistic polynomial-time adversary \mathcal{S} for the ideal model such that for every $I \subseteq [n]$, every balanced vector $\bar{x} \in (\{0, 1\}^*)^n$, and every auxiliary input $z \in \{0, 1\}^*$:*

$$\{\text{IDEALC}_{f, \mathcal{S}(z), I}^\epsilon(\bar{x}, 1^\kappa)\}_{\kappa \in \mathbb{N}} \stackrel{c}{\equiv} \{\text{REAL}_{\Pi, \mathcal{A}(z), I}(\bar{x}, 1^\kappa)\}_{\kappa \in \mathbb{N}}$$

Notice that the definition of the ideal world given above differs from the original definition of Aumann and Lindell in four aspects. First, we add the support of functions with no private inputs from the parties to model input-independent functionalities. In this case, the parties send `ok` instead of their inputs to the trusted party. Second, whenever a corrupted party aborts, the trusted party sends `abort` to all honest parties. Note that this message does not include the index of the aborting party which differs from the original model. The security notion of *identifiable abort* [IOZ14], where the aborting party is identified, is an independent research area, and is not achieved by our compiler. Third, we allow a corrupted party to abort after `undetected` cheating, which does not weaken the security guarantees.

Finally, we allow the adversary to learn the output of the function f before it decides to cheat or to act honestly. In the original notion the adversary has

to make this decision without seeing the potential output. Although this modification gives the adversary additional power, it captures the real world more reliably in regard to standalone input-independent protocols.

Covert security is typically achieved by executing several semi-honest instances and checking some of them via cut-and-choose while utilizing an unchecked instance for the actual output generation. The result of the semi-honest instances is often an input-independent precomputation in the form of correlated randomness, e.g., a garbled circuit or multiplication triples, which is consumed in a maliciously secure input-dependent online phase, e.g., the circuit evaluation or a SPDZ-style [DKL⁺13] online phase. Typically, the precomputation is explicitly designed not to leak any information about the actual output of the online phase, e.g., a garbled circuit obfuscates the actual circuit gate tables and multiplication triples are just random values without any relation to the output or even the function computed in the online phase. Thus, in such protocols, the adversary does not learn anything about the output when executing the semi-honest instances and therefore when deciding to cheat, which makes the original notion of covert security realistic for such input-dependent protocols.

However, if covert security is applied to the standalone input-independent precomputation phase, as done by our compiler, the actual output is the correlated randomness provided by one of the semi-honest instances. Hence, the adversary learns potential outputs when executing the semi-honest instances. Considering a rushing adversary that learns the output of a semi-honest instance first and still is capable to cheat with its last message, the adversary can base its decision to cheat on potential outputs of the protocol. Although this scenario is simplified and there is often a trade-off between output determination and cheating opportunities, the adversary potentially learns something about the output before deciding to cheat. This is a power that the adversary might have in all cut-and-choose-based protocols that do not further process the output of the semi-honest instances, also in the input-independent covert protocols compiled by Damgård et al. [DOS20].

Additionally, as we have highlighted above, the result of the precomputation typically does not leak any information about an input-dependent phase which uses this precomputation. Hence, in such offline-online protocols, the adversary has only little benefit of seeing the result of the precomputation before deciding to cheat or to act honestly.

Instead of adapting the notion of covert security, we could also focus on protocols that first obfuscate the output of the semi-honest instances, e.g., by secret sharing it, and then de-obfuscate the output in a later stage. However, this restricts the compiler to a special class of protocols but has basically the same effect. If we execute such a protocol with our notion of security up to the obfuscation stage but without de-obfuscating, the adversary learns the potential output, that is just some obfuscated output and therefore does not provide any benefit to the adversary's cheat decision. Next, we only have to ensure that the de-obfuscating is done in a malicious or covert secure way, which can be achieved,

e.g., by committing to all output shares after the semi-honest instances and then open them when the cut-and-choose selection is done.

For the above reasons, we think it is a realistic modification to the covert notion to allow the adversary to learn the output of the function f before she decides to cheat or to act honestly. Note that the real-world adversary in cut-and-choose-based protocols does only see a list of potential outputs but the ideal-world adversary receives a single output which is going to be the protocol output if the adversary does not cheat or abort. However, we have chosen to be more generous to the adversary and model the ideal world like this in order to keep it simpler and more general. For the same reason we ignore the trade-off between output determination and cheating opportunities observed in real-world protocols.

In the rest of this work, we denote the trusted party computing function f in the ideal-world description by \mathcal{F}_{Cov} .

2.2 Covert Security with Public Verifiability

As discussed in the introduction Asharov and Orlandi introduced to notion of *covert security with ϵ -deterrent and public verifiability* (PVC) in the two-party setting [AO12]. We give an extension of their formal definition to the multi-party setting in the following.

In addition to the covert secure protocol Π , we define two algorithms **Blame** and **Judge**. **Blame** takes as input the view of an honest party P_i after P_i outputs corrupted_j in the protocol execution for $j \in I$ and returns a certificate **Cert**, i.e., $\text{Cert} := \text{Blame}(\text{view}_i)$. The **Judge**-algorithm takes as input a certificate **Cert** and outputs the identity id_j if the certificate is valid and states that party P_j behaved maliciously; otherwise, it returns **none** to indicate that the certificate was invalid.

Moreover, we require that the protocol Π is slightly adapted such that an honest party P_i computes $\text{Cert} = \text{Blame}(\text{view}_i)$ and broadcasts it after cheating has been detected. We denote the modified protocol by Π' . Notice that due to this change, the adversary gets access to the certificate. By requiring simulatability, it is guaranteed that the certificate does not reveal any private information.

We now continue with the definition of covert security with ϵ -deterrent and public verifiability in the multi-party case.

Definition 2 (Covert security with ϵ -deterrent and public verifiability in the multi-party case (PVC-MPC)). *Let f, Π', Blame , and **Judge** be as above. The triple $(\Pi', \text{Blame}, \text{Judge})$ securely computes f in the presence of covert adversaries with ϵ -deterrent and public verifiability if the following conditions hold:*

1. *(Simulatability) The protocol Π' securely computes f in the presence of covert adversaries with ϵ -deterrent according to the strong explicit cheat formulation (see Definition 1).*

2. (*Public Verifiability*) For every PPT adversary \mathcal{A} corrupting parties P_i for $i \in I \subseteq [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0, 1\}^*)^{n+1}$ the following holds:
 If $\text{OUTPUT}_j(\text{REAL}_{\Pi, \mathcal{A}(z), I}(\bar{x}, 1^\kappa)) = \text{corrupted}_i$ for $j \in [n] \setminus I$ and $i \in I$ then:

$$\Pr[\text{Judge}(\text{Cert}) = \text{id}_i] > 1 - \mu(n),$$

where Cert is the output certificate of the honest party P_j in the execution.

3. (*Defamation Freeness*) For every PPT adversary \mathcal{A} corrupting parties P_i for $i \in I \subseteq [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0, 1\}^*)^{n+1}$ and all $j \in [n] \setminus I$:

$$\Pr[\text{Cert}^* \leftarrow \mathcal{A}; \text{Judge}(\text{Cert}^*) = \text{id}_j] < \mu(n).$$

3 Preliminaries

3.1 Communication Model & Notion of Time

We assume the existence of authenticated channels between every pair of parties. Further, we assume synchronous communication between all parties participating in the protocol execution. This means the computation proceeds in rounds, where each party is aware of the current round. All messages sent in one round are guaranteed to arrive at the other parties at the end of this round. We further consider rushing adversaries which in each round are able to learn the messages sent by other parties before creating and sending their own messages. This allows an adversary to create messages depending on messages sent by other parties in the same round.

We denote the time for a single communication round by T_c . In order to model the time, it takes to compute algorithms, we use the approach presented by Wesolowski [Wes19]. Suppose the adversary works in computation model \mathcal{M} . The model defines a cost function C and a time-cost function T . $C(\mathcal{A}, x)$ denotes the overall cost to execute algorithm \mathcal{A} on input x . Similar, the time-cost function $T(\mathcal{A}, x)$ abstracts the notion of time of running $\mathcal{A}(x)$. Considering circuits as computational model, one may consider the cost function denoting the overall number of gates of the circuit and the time-cost function being the circuit's depth.

Let \mathcal{S} be an algorithm that for any RSA modulus N generated with respect to the security parameter κ on input N and some element $g \in \mathbb{Z}_N$ outputs the square of g . We define the time-cost function $\delta_{\text{sq}}(\kappa) = T(\mathcal{S}, (N, g))$, i.e., the time it takes for the adversary to compute a single squaring modulo N .

3.2 Verifiable Time-Lock Puzzle

Time-lock puzzles (TLP) provide a mean to encrypt messages to the future. The message is kept secret at least for some predefined time. The concept of a time-lock puzzle was first introduced by Rivest et al. [RSW96] presenting an elegant

construction using sequential squaring modulo a composite integer $N = p \cdot q$, where p and q are primes. The puzzle is some $x \in \mathbb{Z}_N^*$ with corresponding solution $y = x^{2^T}$. The conjecture about this construction is that it requires T sequential squaring to find the solution. Based on the time to compute a single squaring modulo N , the hardness parameter \mathcal{T} denotes the amount of time required to decrypt the message. (See Section 3.1 for a notion of time.)

We extend the notion of time-lock puzzle by a verifiability notion. This property allows a party who solved a puzzle to generate a proof which can be efficiently verified by any third party. Hence, a solver is able to create a verifiable statement about the solution of a puzzle. Boneh et al. [BBBF18] introduced the notion of verifiable delay functions (VDF). Similar to solving a TLP, the evaluation of a VDF on some input x takes a predefined number of sequential steps. Together with the output y , the evaluator obtains a short proof π . Any other party can use π to verify that y was obtained by evaluating the VDF on input x . Besides the sequential evaluation, a VDF provides no means to obtain the output more efficiently. Since we require a primitive that allows a party using some trapdoor information to perform the operation more efficiently, we cannot use a VDF but start with a TLP scheme and add verifiability using known techniques.

We present a definition of verifiable time-lock puzzles. We include a setup algorithm in the definition which generates public parameters required to efficiently construct a new puzzle. This way, we separate expensive computation required as a one-time setup from the generation of puzzles.

Definition 3. *Verifiable time-lock puzzle (VTLP)* A verifiable time-lock puzzle scheme over some finite domain \mathcal{S} consists of four probabilistic polynomial-time algorithms (TL.Setup, TL.Generate, TL.Solve, TL.Verify) defined as follows.

- $(pp) \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T})$ takes as input the security parameter 1^λ and a hardness parameter \mathcal{T} , and outputs public parameter pp .
- $p \leftarrow \text{TL.Generate}(pp, s)$ takes as input public parameters pp and a solution $s \in \mathcal{S}$ and outputs a puzzle p .
- $(s, \pi) \leftarrow \text{TL.Solve}(pp, p)$ is a deterministic algorithm that takes as input public parameters pp and a puzzle p and outputs a solution s and a proof π .
- $b := \text{TL.Verify}(pp, p, s, \pi)$ is a deterministic algorithm that takes as input public parameters pp , a puzzle p , a solution s , and a proof π and outputs a bit b , with $b = 1$ meaning valid and $b = 0$ meaning invalid. Algorithm TL.Verify must run in total time polynomial in $\log \mathcal{T}$ and λ .

We require the following properties of a verifiable time-lock puzzle scheme.

Completeness For all $\lambda \in \mathbb{N}$, for all \mathcal{T} , for all $pp \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T})$, and for all s , it holds that

$$(s, \cdot) \leftarrow \text{TL.Solve}(\text{TL.Generate}(pp, s)).$$

Correctness For all $\lambda \in \mathbb{N}$, for all \mathcal{T} , for all $pp \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T})$, for all s , and for all $p \leftarrow \text{TL.Generate}(pp, s)$, if $(s, \pi) \leftarrow \text{TL.Solve}(p)$, then

$$\text{TL.Verify}(pp, p, s, \pi) = 1.$$

Soundness For all $\lambda \in \mathbb{N}$, for all \mathcal{T} , and for all PPT algorithms \mathcal{A}

$$\Pr \left[\begin{array}{l} \text{TL.Verify}(pp, p', s', \pi') = 1 \\ s' \neq s \end{array} \middle| \begin{array}{l} pp \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T}) \\ (p', s', \pi') \leftarrow \mathcal{A}(1^\lambda, pp, \mathcal{T}) \\ (s, \cdot) \leftarrow \text{TL.Solve}(pp, p') \end{array} \right] \leq \text{negl}(\lambda)$$

Security A VTLP scheme is secure with gap $\epsilon < 1$ if there exists a polynomial $\tilde{\mathcal{T}}(\cdot)$ such that for all polynomials $\mathcal{T}(\cdot) \geq \tilde{\mathcal{T}}(\cdot)$ and every polynomial-size adversary $(\mathcal{A}_1, \mathcal{A}_2) = \{(\mathcal{A}_1, \mathcal{A}_2)_\lambda\}_{\lambda \in \mathbb{N}}$ where the depth of \mathcal{A}_2 is bounded from above by $\mathcal{T}^\epsilon(\lambda)$, there exists a negligible function $\mu(\cdot)$, such that for all $\lambda \in \mathbb{N}$ it holds that

$$\Pr \left[b \leftarrow \mathcal{A}_2(pp, p, \tau) \middle| \begin{array}{l} (\tau, s_0, s_1) \leftarrow \mathcal{A}_1(1^\lambda) \\ pp \leftarrow \text{TL.Setup}(1^\lambda, \mathcal{T}(\lambda)) \\ b \stackrel{\$}{\leftarrow} \{0, 1\} \\ p \leftarrow \text{TL.Generate}(pp, s_b) \end{array} \right] \leq \frac{1}{2} + \mu(\lambda)$$

and $(s_0, s_1) \in \mathcal{S}^2$.

Although our compiler can be instantiated with any TLP scheme satisfying Definition 3, we present a concrete construction based on the RSW time-lock puzzle [RSW96]. We leave it to further research to investigate if a time-lock puzzle scheme matching our requirements, i.e., verifiability and efficient puzzle generation, can be constructed based on hidden order groups with public setup such as ideal class groups of imaginary quadratic fields [BW88] or Jacobians of hyperelliptic curves [DG20]. Due to the public setup, such constructions might be more efficient than our RSW-based solution.

In order to make the decrypted value verifiable we integrate the generation of a proof as introduced by Wesolowski [Wes19] for verifiable delay functions. The technique presented by Wesolowski provides a way to generate a small proof which can be efficiently verified. However, proof generation techniques from other verifiable delay functions, e.g., presented by Pietrzak [Pie19] can be used as well. The approach of Wesolowski utilizes a function bin, which maps an integer to its binary representation, and a hash function H_{prime} that maps any string to an element of $\text{Primes}(2k)$. The set $\text{Primes}(2k)$ contains the first 2^{2k} prime numbers, where k denotes the security level (typically 128, 192 or 256).

The TL.Setup -algorithm takes the security and hardness parameter and outputs public parameter. This includes an RSA modulus of two strong primes, the number of sequential squares corresponding to the hardness parameter, and a base puzzle. The computation can be executed efficiently if the prime numbers are known. Afterwards, the primes are not needed anymore and can be thrown away. Note that any party knowing the factorization of the RSA modulus can efficiently solve puzzles. Hence, the TL.Setup -algorithm should be executed in a trusted way.

The TL.Generate -algorithm allows any party to generate a time-lock puzzle over some secret s . In the construction given below, we assume s to be an element in \mathbb{Z}_N^* . However, one can use a hybrid approach where the secret is encrypted

with some symmetric key which is then mapped to an element in \mathbb{Z}_N^* . This allows the generator to time-lock large secrets as well. Note that the puzzle generation can be done efficiently and does not depend on the hardness parameter \mathcal{T} .

The `TL.Solve`-algorithm solves a time-lock puzzle p by performing sequential squaring, where the number of steps depend on the hardness parameter \mathcal{T} . Along with the solution, it outputs a verifiable proof π . This proof is used as additional input to the `TL.Verify`-algorithm outputting true if the given secret was time-locked by the given puzzle.

We state the formal definition of our construction next.

Construction Verifiable Time-Lock Puzzle

`TL.Setup`($1^\lambda, \mathcal{T}$):

- Sample two strong primes (p, q) and set $N := p \cdot q$.
- Set $\mathcal{T}' := \mathcal{T} / \delta_{\text{sq}}(\lambda)$.
- Sample uniform $\tilde{g} \xleftarrow{\$} \mathbb{Z}_N^*$ and set $g := -\tilde{g}^2 \pmod{N}$.
- Compute $h := g^{2^{\mathcal{T}'}}$, which can be optimized by reducing $2^{\mathcal{T}'}$ module $\phi(N)$ first.
- Set $Z := (g, h)$.
- Output (\mathcal{T}', N, Z) .

`TL.Generate`(pp, s):

- Parse $pp := (\mathcal{T}', N, Z)$ and $Z := (g, h)$.
- Sample uniform $r \xleftarrow{\$} \{1, \dots, N^2\}$.
- Compute $g^* := g^r$ and $h^* := h^r$.
- Set $c^* := h^* \cdot s \pmod{N}$.
- Output $p := (g^*, c^*)$.

`TL.Solve`(pp, p):

- Parse $pp := (\mathcal{T}', N, Z)$ and $p := (g^*, c^*)$.
- Compute $h := g^{*2^{\mathcal{T}'}} \pmod{N}$ by repeated squaring.
- Compute $s := \frac{c^*}{h} \pmod{N}$ as the solution.
- Compute $\ell = H_{\text{prime}}(\text{bin}(g^*) || * || \text{bin}(s)) \in \text{Primes}(2k)$ as the challenge.
- Compute $\pi = g^{* \lfloor 2^{\mathcal{T}' / \ell} \rfloor}$ as the proof.
- Output (s, π) .

`TL.Verify`(pp, p, s, π):

- Parse $pp := (\mathcal{T}', N, Z)$.
- Parse $p := (g^*, c^*)$.
- Compute $\ell = H_{\text{prime}}(\text{bin}(g^*) || * || \text{bin}(s)) \in \text{Primes}(2k)$ as the challenge.
- Compute $r = 2^{\mathcal{T}' \pmod{\ell}}$.
- Compute $h' = \pi^\ell g^{*r}$.
- Compute $s' := \frac{c^*}{h'}$.
- If $s = s'$, output 1, otherwise output 0.

The security of the presented construction is based on the conjecture that it requires \mathcal{T}' sequential squarings to solve a puzzle. Moreover, the soundness of the proof generation is based on the number-theoretic assumption that it is hard to find the ℓ -th root modulo an RSA modulus N of an integer $x \notin \{-1, 0, +1\}$ where ℓ is uniformly sampled from $\text{Primes}(2k)$ and the factorization of N is unknown. See [Wes19] for a detailed description of the security assumption.

3.3 Commitment

Our protocol makes use of an extractable commitment scheme which is *computationally binding and hiding*. For ease of description, we assume the scheme to be non-interactive. We will use the notation $(c, d) \leftarrow \text{Commit}(m)$ to commit to message m , where c is the commitment value and d denotes the decommitment or opening value. Similarly, we use $m' \leftarrow \text{Open}(c, d)$ to open commitment c with opening value d to $m' = m$ or $m' = \perp$ in case of incorrect opening. The extractability property allows the simulator to extract the committed message m and the opening value d from the commitment c by using some trapdoor information.

Such a scheme can be implemented in the random oracle model by defining $\text{Commit}(x) = H(i, x, r)$ where i is the identity of the committer, $H : \{0, 1\}^* \rightarrow \{0, 1\}^{2\kappa}$ is a random oracle and $r \xleftarrow{\$} \{0, 1\}^\kappa$.

3.4 Signature Scheme

We use a signature scheme ($\text{Gen}, \text{Sign}, \text{Verify}$) that is *existentially unforgeable under chosen-message attacks*. Before the start of our protocol, each party executes the Gen -algorithm to obtain a key pair (pk, sk) . While the secret key sk is kept private, we assume that each other party is aware of the party's public key pk .

3.5 Semi-Honest Base Protocol

Our compiler is designed to transform a semi-honest secure n -party protocol with no private input tolerating $n - 1$ corruptions, Π_{SH} , that computes a probabilistic function $(y^1, \dots, y^n) \leftarrow f()$, where y^i is the output for party P_i , into a publicly verifiable covert protocol, Π_{PVC} , that computes the same function. In order to compile Π_{SH} , it is necessary that all parties that engage in the protocol Π_{SH} receive a protocol transcript, which is the same if all parties act honestly. This means that there needs to be a fixed ordering for the sent messages and that each message needs to be sent to all involved parties ⁴.

We stress that any protocol can be adapted to fulfill the compilation requirements. Adding a fixed order to the protocol messages is trivial and just a matter of specification. Furthermore, parties can send all of their outgoing messages to all other parties without harming the security. This is due to the fact, that the

⁴ This requirement is inherent to all known publicly verifiable covert secure protocols.

protocol tolerates $n - 1$ corruptions which implies that the adversary is allowed to learn all messages sent by the honest party anyway. Note that the transferred messages do not need to be securely broadcasted, because our compiler requires the protocol to produce a consistent transcript only if all parties act honestly.

3.6 Coin Tossing Functionality

We utilize a maliciously secure coin tossing functionality $\mathcal{F}_{\text{coin}}$ parameterized with the security parameter κ and the number of parties n . The functionality receives ok_i from each party P_i for $i \in [n]$ and outputs a random κ -bit string $\text{seed} \xleftarrow{\$} \{0, 1\}^\kappa$ to all parties.

Functionality $\mathcal{F}_{\text{coin}}$

Inputs: Each party P_i with $i \in [n]$ inputs ok_i .

- Sample $\text{seed} \xleftarrow{\$} \{0, 1\}^\kappa$.
- Send seed to \mathcal{A} .
 - If \mathcal{A} returns **abort**, send **abort** to all honest parties and stop.
 - Otherwise, send seed to all honest parties.

3.7 Puzzle Generation Functionality

The maliciously secure puzzle generation functionality \mathcal{F}_{PG} is parameterized with the computational security parameter κ , the number of involved parties n , the cut-and-choose parameter t and public TLP parameters pp . It receives a coin share r^i , a puzzle randomness share u^i , and the seed-share decommitments for all instances $\{d_j^i\}_{j \in [t]}$ as input from each party P_i . \mathcal{F}_{PG} calculates the random coin r and the puzzle randomness u using the shares of all parties. Then, it generates a time-lock puzzle p of r and all seed-share decommitments except the ones with index r . In the first output round it sends p to all parties. In the second output round it reveals the values locked within p to all parties. As we assume a rushing adversary, \mathcal{A} receives the outputs first in both rounds and can decide if the other parties should receive the outputs as well.

The functionality \mathcal{F}_{PG} can be instantiated with a general purpose maliciously secure MPC-protocol such as the ones specified by [DKL⁺13] or [YWZ20].

Functionality \mathcal{F}_{PG}

Inputs: Each party P_i with $i \in [n]$ inputs $(r^i, u^i, \{d_j^i\}_{j \in [t]})$, where $r^i \in [t]$, $u^i \in \{0, 1\}^\kappa$, and $d_j^i \in \{0, 1\}^\kappa$.

- Compute $r := \sum_{i=1}^n r^i \bmod t$ and $u := \bigoplus_{i=1}^n u^i$.
- Generate puzzle $p \leftarrow \text{TL.Generate}(pp, (r, \{d_j^i\}_{i \in [n], j \in [t] \setminus \{r\}}))$ using randomness u .
- Send p to \mathcal{A} .

- If \mathcal{A} returns **abort**, send **abort** to all honest parties and stop.
- Otherwise, send p to all honest parties.⁵
- Upon receiving **continue** from each party, send $(r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r})$ to \mathcal{A} .
 - If \mathcal{A} returns **abort** or some party does not send **continue**, send **abort** to all honest parties and stop.
 - Otherwise, send $(r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r})$ to all honest parties.

4 PVC Compiler

In the following, we present our compiler for multi-party protocols with no private input from semi-honest to publicly verifiable covert security. We start with presenting a distributed seed computation which is used as subprotocol in our compiler. Next, we state the detailed description of our compiler. Lastly, we provide information about the **Blame**- and **Judge**-algorithm required by the notion of publicly verifiable covert security.

4.1 Distributed Seed Computation

The execution of the semi-honest protocol instances Π_{SH} within our PVC compiler requires each party to use a random tape that is uniform at random. In order to ensure this requirement, the parties execute several instances of a distributed seed computation subprotocol Π_{SG} at the beginning. During this subprotocol, each party P_h selects a uniform κ -bit string as private seed share $\text{seed}^{(1,h)}$. Additionally, P_h and all other parties get uniform κ -bit strings $\{\text{seed}^{(2,i)}\}_{i \in [n]}$, which are the public seed shares of all parties. The randomness used by P_h in the semi-honest protocol will be derived from $\text{seed}^h := \text{seed}^{(1,h)} \oplus \text{seed}^{(2,h)}$. This way seed^h is distributed uniformly. Note that if protocol Π_{SH} is semi-malicious instead of semi-honest secure then each party may choose the randomness arbitrarily and there is no need to run the seed generation.

As the output, party P_h obtains its own private seed, commitments to all private seeds, a decommitment for its own private seed, and all public seed shares. We state the detailed protocol steps next. The protocol is executed by each party P_h , parameterized with the number of parties n and the security parameter κ .

Protocol Π_{SG}

- (a) **Commit-phase**
Party P_h chooses a uniform κ -bit string $\text{seed}^{(1,h)}$, sets $(c^h, d^h) \leftarrow \text{Commit}(\text{seed}^{(1,h)})$, and sends c^h to all parties.
 - (b) **Public coin-phase**
For each $i \in [n]$, party P_h sends **ok** to $\mathcal{F}_{\text{coin}}$ and receives $\text{seed}^{(2,i)}$.
- Output**

⁵ The honest parties receive p or **abort** in the same communication round as \mathcal{A} .

If P_h has not received all messages in the expected communication rounds or any $\text{seed}^{(2,i)} = \perp$, it sends **abort** to all parties and outputs **abort**. Otherwise, it outputs $(\text{seed}^{(1,h)}, d^h, \{\text{seed}^{(2,i)}, c^i\}_{i \in [n]})$.

4.2 The PVC Compiler

Starting with a n -party semi-honest secure protocol Π_{SH} we compile a publicly verifiable covert secure protocol Π_{PVC} . The compiler works for protocols that receive no private input.

The compiler uses a signature scheme, a verifiable time-lock puzzle scheme, and a commitment scheme as building blocks. Moreover, the communication model is as defined in Section 3.1. We assume each party generated a signature key pair (sk, pk) and all parties know the public keys of the other parties. Furthermore, we suppose the setup of the verifiable time-lock puzzle scheme TL.Setup was executed in a trusted way beforehand. This means in particular that all parties are aware of the public parameters pp . We stress that this setup needs to be executed once and may be used by many protocol executions. The hardness parameter \mathcal{T} used as input to the TL.Setup -algorithm needs to be defined as $\mathcal{T} > 2 \cdot T_c$, where T_c denotes the time for a single communication round (see Section 3.1). In particular, the hardness parameter is independent of the complexity of Π_{SH} .

From a high-level perspective, our compiler works in five phases. At the beginning, all parties jointly execute the seed generation to set up seeds from which the randomness in the semi-honest protocol instances is derived. Second, the parties execute t instances of the semi-honest protocol Π_{SH} . By executing several instances, the parties' honest behavior can be later on checked in all but one instance. Since checking reveals the confidential outputs of the other parties, there must be one instance that is unchecked. The index of this one is jointly selected in a random way in the third phase. Moreover, publicly verifiable evidence is generated such that an honest party can blame any malicious behavior afterwards. To this end, we use the puzzle generation functionality \mathcal{F}_{PG} to generate a time-lock puzzle first. Next, each party signs all information required for the other parties to blame this party. In the fourth phase, the parties either honestly reveal secret information for all but one semi-honest execution or abort. In case of abort, the honest parties execute the fifth phase. By solving the time-lock puzzle, the honest parties obtain the required information to create a certificate about malicious behavior. Since this phase is only required to be executed in case any party aborted before revealing the information, we call this the pessimistic case. We stress that no honest party is required to solve a time-lock puzzle in case all parties behave honestly.

A corrupted party may cheat in two different ways in the compiled protocol. Either the party inputs decommitment values into the puzzle generation functionality which open the commitments created during the seed generation to \perp or the party misbehaved in the execution of Π_{SH} . The later means that a

party uses different randomness than derived from the seeds generated at the beginning.

The first cheat attempt may be detected in two ways. In the optimistic execution, all parties receive the inputs to \mathcal{F}_{PG} and can verify that opening the commitments is successful. In the pessimistic case, solving the time-lock puzzle reveals the input to \mathcal{F}_{PG} . Since we do not want the Judge to solve the puzzle itself, we provide a proof along with the solution of the time-lock puzzle. To this end, we require a verifiable time-lock puzzle as modeled in Section 3. Even in the optimistic case, if an honest party detects cheating, the time-lock puzzle needs to be solved in order to generate a publicly verifiable certificate.

If all decommitments open the commitments successfully, an honest party can recompute the seeds used by all other parties in an execution of Π_{SH} and re-run the execution. The resulting transcript is compared with the one signed by all parties beforehand. In case any party misbehaved, a publicly verifiable certificate can be created. For the sake of exposition, we compress the detection of malicious behavior and the generation of the certificate into the **Blame**-algorithm.

The protocol defined as follows is executed by each honest party P_h .

Protocol Π_{PVC}

Public input: All parties agree on $\kappa, n, t, \Pi_{\text{SH}}$ and pp and know all parties' public keys $\{pk_i\}_{i \in [n]}$.

Private input: P_h knows its own secret key sk_h .

Distributed seed computation:

We abuse notation here and assume that the parties execute the seed generation protocol from above.

1. For each instance $j \in [t]$ party P_h interacts with all other parties to receive

$$(\text{seed}_j^{(1,h)}, d_j^h, \{\text{seed}_j^{(2,i)}, c_j^i\}_{i \in [n]}) \leftarrow \Pi_{\text{SG}}$$

and computes $\text{seed}_j^h := \text{seed}_j^{(1,h)} \oplus \text{seed}_j^{(2,h)}$.

Semi-honest protocol execution:

2. Party P_h engages in t instances of the protocol Π_{SH} with all other parties. In the j -th instance, party P_h uses randomness derived from seed_j^h and receives a transcript and output:

$$(\text{trans}_j, y_j^h) \leftarrow \Pi_{\text{SH}}.$$

Create publicly verifiable evidence:

3. Party P_h samples a coin share $r^h \xleftarrow{\$} [t]$, a randomness share $u^h \xleftarrow{\$} \{0, 1\}^\kappa$, sends the message $(r^h, u^h, \{d_j^h\}_{j \in [t]})$ to \mathcal{F}_{PG} and receives time-lock puzzle p as response.
4. For each $j \in [t]$, Party P_h creates a signature $\sigma_j^h \leftarrow \text{Sign}_{sk_h}(\text{data}_j)$, where the signed data is defined as

$$\text{data}_j := (h, j, \{\text{seed}_j^{(2,i)}\}_{i \in [n]}, \{c_j^i\}_{i \in [n]}, p, \text{trans}_j).$$

P_h broadcasts its signatures and verifies the received signatures.

Optimistic case:

5. If any of the following cases happens
 - P_h has not received valid messages in the first protocol steps in the expected communication round.
 - \mathcal{F}_{PG} returned **abort**, or
 - any other party has sent **abort**
 party P_h broadcasts and outputs **abort**.
6. Otherwise, P_h sends **continue** _{h} to \mathcal{F}_{PG} , receives $(r, \{d_j^{*i}\}_{i \in [n], j \in [t] \setminus r})$ as response and calculates

$$(m, \text{cert}) := \text{Blame}(\text{view}^h)$$

where view^h is the view of P_h .

If $\text{cert} \neq \perp$, broadcast **cert** and output **corrupted** _{m} . Otherwise, P_h outputs y_r^h .

Pessimistic case:

7. If \mathcal{F}_{PG} returned **abort** in step 6, P_h solves the time-lock puzzle

$$((r, \{d_j^{*i}\}_{i \in [n], j \in [t] \setminus r}), \pi) := \text{TL.Solve}(pp, p)$$

and calculates

$$(m, \text{cert}) := \text{Blame}(\text{view}^h)$$

where view^h is the view of P_h .

If $\text{cert} \neq \perp$, broadcast **cert** and output **corrupted** _{m} . Otherwise, output **abort**.

4.3 Blame-Algorithm

Our PVC compiler uses an algorithm **Blame** in order to verify the behavior of all parties in the opened protocol instances and to generate a certificate of misbehavior if cheating has been detected. It takes the view of a party as input and outputs the index of the corrupted party in addition to the certificate. If there are several malicious parties the algorithm selects the one with the minimal index.

Algorithm Blame

On input the view **view** of a party which contains:

- public parameters (n, t)
- public seed shares $\{\text{seed}_j^{(2,i)}\}_{i \in [n]}$
- shared coin r
- private seed share commitments and decommitments $\{c_j^i, d_j^i\}_{i \in [n], j \in [t] \setminus r}$
- additional certificate information
 $(\{\text{pk}_j\}_{i \in [n]}, \{\text{data}_j\}_{j \in [t]}, \pi, \{\sigma_j^i\}_{i \in [n], j \in [t]})$

do:

1. Calculate $\text{seed}_j^{(1,i)} := \text{Open}(c_j^i, d_j^i)$ for each $i \in [n], j \in [t] \setminus r$.
2. Let $M_1 := \{(i, j) \in ([n], [t] \setminus r) : \text{seed}_j^{(1,i)} = \perp\}$. If $M_1 \neq \emptyset$, choose the tuple $(m, l) \in M_1$ with minimal m and l , prioritized by m , compute $(\cdot, \pi) := \text{TL.Solve}(pp, p)$, if $\pi = \perp$, set $\text{cert} := (\text{pk}_m, \text{data}_j, \pi, r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r}, \sigma_l^m)$ and output (m, cert) .
3. Set $\text{seed}_j^i := \text{seed}_j^{(1,i)} \oplus \text{seed}_j^{(2,i)}$ for all $i \in [n]$ and $j \in [t] \setminus r$.
4. Re-run Π_{SH} for all $j \in [t] \setminus r$ by simulating the view of all other parties: In the j -th instance simulate all parties P_i with randomness seed_j^i for $i \in [n]$ and receive (trans'_j, \cdot) .
5. Let $M_2 := \{j \in [t] \setminus r : \text{trans}'_j \neq \text{trans}_j\}$. If $M_2 \neq \emptyset$, determine the minimal index m such that P_m is the first party that has deviated from the protocol description in an instance $l \in M_2$. If P_m has deviated from the protocol description in several instances $l \in M_2$, choose the smallest such l . Then, set $\text{cert} := (\text{pk}_m, \text{data}_l, \{d_l^i\}_{i \in [n]}, \sigma_l^m)$ and output (m, cert) .
6. Output $(0, \perp)$.

4.4 Judge-Algorithm

The Judge-algorithm receives the certificate and outputs either the identity of the corrupted party or \perp . The execution of this algorithm requires no interaction with the parties participating in the protocol execution. Therefore, it can also be executed by any third party which possesses a certificate cert . If the output is pk_m for $m \in [n]$, the executing party is convinced that party P_m misbehaved during the protocol execution. The Judge-algorithm is parameterized with n, t, pp , and Π_{SH} .

Algorithm Judge(cert)

Inconsistency certificate:

On input $\text{cert} = (\text{pk}_m, \text{data}, \pi, r, \{d_j^i\}_{i \in [n], j \in [t] \setminus r}, \sigma_l^m)$ do:

- If $\text{Verify}_{\text{pk}_m}(\text{data}; \sigma_l^m) = \perp$, output \perp .
- Parse data to $(m, l, \cdot, \{c_i^i\}_{i \in [n]}, p, \cdot)$.
- If $\text{TL.Verify}(pp, p, (r, \{d_j^i\}_{i, j}), \pi) = 0$ output \perp .
- If $r = l$, output \perp .
- If $\text{Open}(c_l^m, d_l^m) \neq \perp$, output \perp . Else output pk_m .

Deviation certificate:

On input $\text{cert} = (\text{pk}_m, \text{data}, \{d_l^i\}_{i \in [n]}, \sigma_l^m)$.

- If $\text{Verify}_{\text{pk}_m}(\text{data}; \sigma_l^m) = \perp$, output \perp .
- Parse data to $(m, l, \{\text{seed}_l^{(2,i)}\}_{i \in [n]}, \{c_i^i\}_{i \in [n]}, \cdot, \text{trans}_l)$.
- Set $\text{seed}_l^{(1,i)} \leftarrow \text{Open}(c_l^i, d_l^i)$ for each $i \in [n]$. If any $\text{seed}_l^{(1,i)} = \perp$, output \perp .
- Set $\text{seed}_l^i := \text{seed}_l^{(1,i)} \oplus \text{seed}_l^{(2,i)}$ for each i .
- Simulate Π_{SH} using the seeds seed_l^i as randomness of party P_i and get result (trans'_l, \cdot) .

- If $\text{trans}'_i = \text{trans}_i$, output \perp . Otherwise, determine the index m' of the first party that has deviated from the protocol description. If $m \neq m'$, output \perp . Otherwise, output pk_m .

Ill formatted: If the cert cannot be parsed to neither of the two above cases, output (\perp) .

5 Security

In this section, we show the security of the compiled protocol described in Section 4. To this end, we state the security guarantee in Theorem 1 and prove its correctness in the following.

Theorem 1. *Let Π_{SH} be a n -party protocol, receiving no private inputs, which is secure against a passive adversary that corrupts up to $n - 1$ parties. Let the signature scheme $(\text{Gen}, \text{Sign}, \text{Verify})$ be existentially unforgeable under chosen-message attacks and let the verifiable time-lock puzzle scheme TL be secure with hardness parameter $\mathcal{T} > 2 \cdot T_c$. Let $(\text{Commit}, \text{Open})$ be an extractable commitment scheme which is computationally binding and hiding. Then protocol Π_{PVC} along with algorithms **Blame** and **Judge** is secure against a covert adversary that corrupts up to $n - 1$ parties with deterrence $\epsilon = 1 - \frac{1}{t}$ and public verifiability according to definition 2 in the $(\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{PG}})$ -hybrid model.⁶*

Proof. We prove security of the compiled protocol Π_{PVC} by showing simulatability, public verifiability, and defamation freeness according to Definition 2 separately.

5.1 Simulatability

In order to prove that Π_{PVC} meets covert security with ϵ -deterrent, we define an ideal-world simulator \mathcal{S} using the adversary \mathcal{A} in a black-box way as a subroutine and playing the role of the parties corrupted by \mathcal{A} when interacting with the ideal covert-functionality \mathcal{F}_{Cov} .

The simulator and the proof that the joint distribution of the honest parties' outputs and the view of \mathcal{A} in the ideal world is computationally indistinguishable from the honest parties' outputs and the view of \mathcal{A} in the real world are given in the full version of the paper.

5.2 Public Verifiability

We first argue that an adversary is not able to perform what we call a *detection dependent abort*. This means that once an adversary learns if its cheating will be detected, it can no longer prevent honest parties from generating a certificate.

⁶ See section 3.1, for details on the notion of time and the communication model.

In order to see this, note that withholding valid signatures by corrupted parties in step 4 results in an abort of all honest parties. In contrast, if all honest parties receive valid signatures from all other parties in step 4, then they are guaranteed to obtain the information encapsulated in the time-lock puzzle, i.e., the coin r and the decommitments of all parties $\{d_j^i\}_{i \in [n], j \in [t] \setminus r}$. Either, all parties jointly trigger the puzzle generation functionality \mathcal{F}_{PG} to output the values or in case any corrupted party aborts, an honest party can solve the time-lock puzzle without interaction. Thus, it is not possible for a rushing adversary that gets the output of \mathcal{F}_{PG} in step 6 first, to prevent the other parties from learning it at some time as well. Moreover, the adversary also cannot extract the values from the puzzles before making the decision if it wants to continue or abort, as the decision has to be made in time smaller than the time required to solve the puzzle. Thus, the adversary's decision to continue or abort is independent from the coin r and therefore independent from the event of being detected or not.

Secondly, we show that the **Judge**-algorithm will accept a certificate, created by an honest party, except with negligible probability. Assume without loss of generality that some malicious party P_m has cheated, cheating has been detected and a certificate (blaming party P_m) has been generated. As we have two types of certificates, we will look at them separately.

If an honest party outputs an *inconsistency certificate*, it has received an inconsistent commitment-opening pair (c_l^m, d_l^m) for some $l \neq r$. The value c_l^m is signed directly by P_m and d_l^m indirectly via the signed time-lock puzzle p . Hence, **Judge** can verify the signatures and detect the inconsistent commitment of P_m as well. Note that due to the verifiability of our time-lock construction, the **Judge**-algorithm does not have to solve the time-lock puzzle itself but just needs to verify a given solution. This enables the algorithm to be executed efficiently.

If an honest party outputs a *deviation certificate*, it has received consistent openings for all $j \neq r$ from all other parties, but party P_m was the first party who deviated from the specification of Π_{SH} in some instance $l \in [t] \setminus r$. Since Π_{SH} requires no input from the parties, deviating from its specification means using different randomness than derived from the seeds generated at the beginning of the compiled protocol. As P_m has signed the transcript trans_l , the private seed-commitments of all parties $\{c_i^j\}_{i \in [n]}$, the public seeds $\{\text{seed}^{(2,i)}\}_{i \in [n]}$, and the certificate contains the valid openings $\{d_i^j\}_{i \in [n]}$, the **Judge**-algorithm can verify that P_m was the first party who misbehaved in instance l the same way the honest party does. Note that it is not necessary for **Judge** to verify that $j \neq r$, because the certificate generating party can only gain valid openings $\{d_i^j\}_{i \in [n]}$ for $j \neq r$.

5.3 Defamation Freeness

Assume, without loss of generality, that some honest party P_h is blamed by the adversary. We show defamation freeness for the two types of certificates separately via a reduction to the security of the commitment scheme, the signature scheme and the time-lock puzzle scheme.

First, assume there is a valid *inconsistency certificate* cert^* blaming P_h . This means that there is a valid signature of P_h on a commitment c_j^{*h} and a time-lock puzzle p^* that has a solution s^* which contains an opening d_j^{*h} such that $\text{Open}(c_j^{*h}, d_j^{*h}) = \perp$ and $j \neq r$. As P_h is honest, P_h only signs a commitment c_j^{*h} which equals the commitment honestly generated by P_h during the seed generation. We call such a c_j^{*h} *correct*. Thus, c_j^{*h} is either correct or the adversary can forge signatures. Similar, P_h does only sign the puzzle p^* received by \mathcal{F}_{PG} . This puzzle is generated on the opening value provided by all parties. Since P_h is honest, correct opening values are inserted. Therefore, the signed puzzle p^* either contains the correct opening value or the adversary can forge signatures. Due to the security guarantees of the puzzle, the adversary has to either provide the correct solution s^* or can break the soundness of the time-lock puzzle scheme. To sum it up, an adversary creating a valid *inconsistency certificate* contradicts to the security assumptions specified in Theorem 1.

Second, assume there is a valid *deviation certificate* cert^* blaming P_h . This means, there is a protocol transcript trans_j^* in which P_h is the first party that has sent a message which does not correspond to the next-message function of Π_{SH} and the randomness, seed_j^h used by the judge to simulate P_h . As P_h is honest, either trans_j^* or seed_j^h needs to be incorrect. Also, P_h does not create a signature for an invalid trans_j^* . Thus, trans_j^* is either correct or the adversary can forge signatures. The seed_j^h is calculated as $\text{seed}_j^h := \text{seed}_j^{(1,h)} \oplus \text{seed}_j^{(2,h)}$. The public seed $\text{seed}_j^{(2,h)}$ is signed by P_h and provided directly. The private seed of P_h is provided via a commitment-opening pair (c_j^h, d_j^h) , where c_j^h is signed by P_h . As above, c_j^h and $\text{seed}_j^{(2,h)}$ are either correct or the adversary can forge signatures. Similar, d_j^h is either correct or the adversary can break the binding property of the commitment scheme. If the certificate contains correct $(\text{trans}_j^*, c_j^h, d_j^h, \text{seed}_j^{(2,h)})$ the certificate is not valid. Thus, when creating an accepting cert^* , the adversary has either broken the signature or the commitment scheme which contradicts to the assumption of Theorem 1. □

6 Evaluation

6.1 Efficiency of our Compiler

In Section 4, we presented a generic compiler for transforming input-independent multi-party computation protocols with semi-honest security into protocols that offer covert security with public verifiability. We elaborate on efficiency parameters of our construction in the following.

The deterrence factor $\epsilon = \frac{t-1}{t}$ only depends on the number of semi-honest protocol executions t . In particular, ϵ is independent of the number of parties. This property allows for achieving the same deterrence factor for a fixed number of semi-honest executions while the number of parties increases. Our compiler

therefore facilitates secure computation with a large number of parties. Furthermore, the deterrence factor grows with the number of semi-honest instances (t), similar to previous work based on cut-and-choose (e.g., [AL07, AO12, DOS20]). Concretely, this means that for only five semi-honest instances, our compiler achieves a cheating detection probability of 80%. Moreover, the semi-honest instances are independent of each other and, hence, can be executed in parallel. This means, that the communication and computation complexity in comparison to a semi-honest protocol increases by factor t . However, our compiler preserves the round complexity of the semi-honest protocol. Hence, it is particularly useful for settings and protocols in which the round complexity constitutes the major efficiency bottleneck. Similarly, the requirement of sending all messages to all parties further increases the communication overhead by a factor of $n - 1$ but does not affect the round complexity. Since this requirement is inherent to all known publicly verifiable covert secure protocols, e.g., [DOS20], these protocols incur a similar communication overhead.

While our compiler requires a maliciously secure puzzle generation functionality, we stress that the complexity of the puzzle generation is independent of the cost of the semi-honest protocol. Therefore, the relative overhead of the puzzle generation shrinks for more complex semi-honest protocols. One application where our result may be particularly useful is for the preprocessing phase of multi-party computation, e.g., protocols for generating garbled circuits or multiplication triples. In such protocols, one can generate several circuits resp. triples that are used in several online instances but require just one puzzle generation.

For the sake of concreteness, we constructed a boolean circuit for the puzzle generation functionality and estimated its complexity in terms of the number of AND-gates. The construction follows a naive design and should not constitute an efficient solution but should give a first impression on the circuit complexity. We present some intuition on how to improve the circuit complexity afterwards.

We utilize the RSW VTLP construction described in Section 3.2 with a hybrid construction, in which a symmetric encryption key is locked within the actual time-lock puzzle and is used to encrypt the actual secret. Note that the RSW VTLP is not optimized for MPC scenarios. Since our compiler can be instantiated with an arbitrary VTLP satisfying Definition 3, any achievements in the area of MPC-friendly TLP can result into an improved puzzle generation functionality for our compiler. To instantiate the symmetric encryption operation, we use the LowMC [ARS⁺15] cipher, an MPC-friendly cipher tailored for boolean circuits.

Let n be the number of parties, t being the number of semi-honest instances, κ denoting the computational security parameter, and N represents the RSA modulus used for the RSW VTLP. We use the notation $|x|$ to denote the bit length of x . The total number of AND-gates of our naive circuit is calculated as follows:

$$\begin{aligned}
& (n-1) \cdot (11|t| + 22|N| + 12) \\
& + nt \cdot (4|t| + 2\kappa + 755) \\
& + 192|N|^3 + 112|N|^2 + 22|N|
\end{aligned}$$

It is easy to see that the number of AND-gates is linear in both n and t . The most expensive part of the puzzle generation is the computation of two exponentiations required for the RSW VTLP, since the number of required AND-gates is cubic in $|N|$ for an exponentiation. However, we can slightly adapt our puzzle generation functionality and protocol to remove these exponentiations from the maliciously secure puzzle generation protocol. For the sake of brevity, we just give an intuition here.

Instead of performing the exponentiations g^u and h^u required for the puzzle creation within the puzzle generation functionality, we let each party P_i input a 0-puzzle consisting of the two values $g_i = g^{u_i}$ and $h_i = h^{u_i}$. The products of all g_i respectively h_i are used as g^* and h^* for the VTLP computation. Since we replace the exponentiations with multiplications, the number of AND-gates is quadratic instead of cubic in $|N|$.

Note that this modification enables a malicious party to modify the resulting puzzle by inputting a non-zero puzzle. Intuitively, the attacker can render the puzzle invalid such that no honest party can create a valid certificate or the puzzle can be modified such that a corrupted party can create a valid certificate defaming an honest party. Concretely, one possible attack is to input inconsistent values g_i and h_i , i.e., to use different exponents for the two exponentiations. As such an attack must be executed without knowledge of the coin r , it is sufficient to detect invalid inputs and consider such behavior as an early abort. To this end, parties have to provide u_i to the puzzle generation functionality and the functionality outputs $u = \sum u_i$, g^* and h^* in the second output round together with the coin and the seed openings. By comparing if $g^* = g^u$ and $h^* = h^u$, each party can check the validity of the puzzle. Finally, we need to ensure that a manipulated puzzle cannot be used to create an inconsistency certificate blaming an honest party. Such false accusation can easily be prevented, e.g., by adding some zero padding to the value inside the puzzle such that any invalid puzzle input renders the whole puzzle invalid.

6.2 Comparison with Prior Work

To the best of our knowledge, our work is the first to provide a fully specified publicly verifiable multi-party computation protocol against covert adversaries. Hence, we cannot compare to existing protocols directly. However, Damgård et al. [DOS20] have recently presented two compilers for constructing publicly verifiable covert secure protocols from semi-honest secure protocols in the two-party setting, one for input-independent and one for input-dependent protocols. For the latter, they provide an intuition on how to extend the compiler to the multi-party case. However, there is no full compiler specification for neither

input-dependent nor input-independent protocols. Still, there exist a natural extension for the input-independent compiler, which we can compare to.

The major difference between our input-independent protocol and their input-independent protocol, is the way the protocols prevent *detection dependent abort*. In the natural extension to Damgård et al. [DOS20], which we call the *watchlist approach* in the following, each party independently selects a subset of instances it wants to check and receives the corresponding seeds via oblivious transfer. The transcript of the oblivious transfer together with the receiver's randomness can be used by the receiver to prove integrity of its watchlist to the judge; similar to the seed commitments and openings used in our protocol. The watchlists are only revealed after each party receives the data required to create a certificate in case of cheating detection, i.e., the signatures by the other parties. Once a party detects which instances are checked, it is too late to prevent the creation of a certificate. Our approach utilizes time-lock puzzles for the same purpose.

In the watchlist approach, all parties have different watchlists. For t semi-honest instances and watchlists of size $s \geq \frac{t}{n}$, there is a constant probability $\Pr[\text{bad}]$ that no semi-honest instance remains unwatched which leads to a failure of the protocol. Thus, parties either need to choose $s < \frac{t}{n}$ and hence $\epsilon = \frac{s}{t} < \frac{1}{n}$ or run several executions of the protocol. For the latter, the probability of a protocol failure $\Pr[\text{bad}]$ and the expected number of protocol runs runs are calculated based on the inclusion-exclusion principle as follows:

$$\begin{aligned} \Pr[\text{bad}] &= 1 - \frac{\sum_{k=1}^t (-1)^{(k-1)} * \binom{t}{k} * (\prod_{j=0}^{s-1} (t-j-k))^n}{\prod_{j=0}^{s-1} (t-j)^n} \\ &= 1 - \sum_{k=1}^t (-1)^{(k-1)} \cdot \binom{t}{k} \cdot \left(\frac{(t-k)! \cdot (t-s)!}{(t-k-s)! \cdot t!} \right)^n \\ \text{runs} &= \Pr[\text{bad}]^{-1} \end{aligned}$$

Setting the watchlist size $s \geq \frac{t}{n}$ such that there is a constant failure probability has the additional drawback that the repetition can be abused to amplify denial-of-service attacks. An adversary can enforce a high failure probability by selecting its watchlists strategically. If $s \geq \frac{t}{(n-1)}$ and $n-1$ parties are corrupted, the adversary can cause an error with probability 1 which enables an infinite DoS-attack.

This restriction of the deterrence factor seems to be a major drawback of the watchlist approach. Although our approach has an additional overhead due to the puzzle generation, which is independent of the complexity of the transformed protocol and thus amortizes over the complexity of the base protocols, it has the benefit that it immediately supports an arbitrary deterrence factor ϵ . This is due to the fact that the hidden shared coin toss determines a single watchlist shared by all parties. In Table 1, we display the maximal deterrence factor of our approach ϵ in comparison to the maximal deterrence factor of the watchlist approach without protocol repetitions ϵ' for different settings. Additionally,

we provide the number of expected runs required to achieve ϵ in the watchlist approach with repetitions.

n	t	Our approach	Watchlist approach		
		ϵ	ϵ'	or	runs
	2	1/2	-		2
2	3	2/3	1/3		3
	10	9/10	4/10		10
	2	1/2	-		4
3	4	3/4	1/4		16
	10	9/10	3/10		100
5	2	1/2	-		16
	6	5/6	1/6		1296

Table 1. Maximal deterrence factor or expected number of runs of the watchlist approach in comparison to our approach.

Acknowledgments

The first, third, and fourth authors were supported by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 – 236615297 (CROSSING Project S7)*, by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*, and by Robert Bosch GmbH, by the Economy of Things Project. The second author was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister’s Office, and by ISF grant No. 1316/18.

References

- [AL07] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *TCC 2007*.
- [AO12] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. *ASIACRYPT 2012*.
- [ARS⁺15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. *EUROCRYPT 2015, Part I*.
- [BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. *CRYPTO 2018, Part I*.

- [Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. *CRYPTO'91*.
- [BGJ⁺16] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. *ITCS 2016*.
- [BW88] Johannes Buchmann and Hugh C. Williams. A key-exchange system based on imaginary quadratic fields. *Journal of Cryptology*, June 1988.
- [DG20] Samuel Dobson and Steven D. Galbraith. Trustless groups of unknown order with hyperelliptic curves. *IACR Cryptol. ePrint Arch. 2020*, 2020.
- [DGN10] Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. From passive to covert security at low cost. *TCC 2010*.
- [DKL⁺13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. *ESORICS 2013*.
- [DOS20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. *CRYPTO 2020, Part II*.
- [DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zarkarias. Multiparty computation from somewhat homomorphic encryption. *CRYPTO 2012*.
- [GMS08] Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. *EUROCRYPT 2008*.
- [GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. *19th ACM STOC 1987*.
- [HKK⁺19] Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wen-jie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. *EUROCRYPT 2019, Part III*.
- [HVW20] Carmit Hazay, Muthuramakrishnan Venkatasubramanian, and Mor Weiss. The price of active security in cryptographic protocols. *EUROCRYPT 2020, Part II*.
- [IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. *CRYPTO 2014, Part II*.
- [IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. *CRYPTO 2008*.
- [KM15] Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. *ASIACRYPT 2015, Part II*.
- [LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. *CRYPTO 2011*.
- [MMV11] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Time-lock puzzles in the random oracle model. *CRYPTO 2011*.
- [MT19] Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. Homomorphic time-lock puzzles and applications. *CRYPTO 2019, Part I*.
- [Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. *ITCS 2019*.
- [RSW96] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology. Laboratory for Computer Science, 1996.
- [Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. *EUROCRYPT 2019, Part III*.

- [WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. *ACM CCS 17*.
- [WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. *ACM CCS 17*.
- [YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. *ACM CCS 2020*.
- [ZDH19] Ruiyu Zhu, Changchang Ding, and Yan Huang. Efficient publicly verifiable 2pc over a blockchain with applications to financially-secure computations. *ACM CCS 2019*.

B. Financially Backed Covert Security

This chapter corresponds to the following publication. The full version is available at [85].

- [86] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Financially Backed Covert Security”. In: *Public-Key Cryptography - PKC 2022 - 25th IACR International Conference on Practice and Theory of Public-Key Cryptography, Virtual Event, March 8-11, 2022, Proceedings, Part II*. 2022, pp. 99–129. **Part of this thesis.**

Financially Backed Covert Security

Sebastian Faust¹, Carmit Hazay², David Kretzler¹, and Benjamin Schlosser¹

¹ Technical University of Darmstadt, Germany

`{first.last}@tu-darmstadt.de`

² Bar-Ilan University, Israel

`carmit.hazay@biu.ac.il`

Abstract. The security notion of *covert security* introduced by Aumann and Lindell (TCC'07) allows the adversary to successfully cheat and break security with a fixed probability $1 - \epsilon$, while with probability ϵ , honest parties detect the cheating attempt. Asharov and Orlandi (ASIACRYPT'12) extend covert security to enable parties to create publicly verifiable evidence about misbehavior that can be transferred to any third party. This notion is called *publicly verifiable covert security* (PVC) and has been investigated by multiple works. While these two notions work well in settings with known identities in which parties care about their reputation, they fall short in Internet-like settings where there are only digital identities that can provide some form of anonymity.

In this work, we propose the notion of *financially backed covert security* (FBC), which ensures that the adversary is financially punished if cheating is detected. Next, we present three transformations that turn PVC protocols into FBC protocols. Our protocols provide highly efficient judging, thereby enabling practical judge implementations via smart contracts deployed on a blockchain. In particular, the judge only needs to non-interactively validate a single protocol message while previous PVC protocols required the judge to emulate the whole protocol. Furthermore, by allowing an interactive punishment procedure, we can reduce the amount of validation to a single program instruction, e.g., a gate in a circuit. An interactive punishment, additionally, enables us to create financially backed covert secure protocols without any form of common public transcript, a property that has not been achieved by prior PVC protocols.

Keywords: Covert Security · Multi-Party Computation (MPC) · Public Verifiability · Financial Punishment

1 Introduction

Secure multi-party computation (MPC) protocols allow a set of parties to jointly compute an arbitrary function f on private inputs. These protocols guarantee privacy of inputs and correctness of outputs even if some of the parties are corrupted by an adversary. The two standard adversarial models of MPC are *semi-honest* and *malicious* security. While semi-honest adversaries follow the protocol description but try to derive information beyond the output from the

interaction, malicious adversaries can behave in an arbitrary way. MPC protocols in the malicious adversary model provide stronger security guarantees at the cost of significantly less efficiency. As a middle ground between good efficiency and high security Aumann and Lindell introduced the notion of *security against covert adversaries* [AL07]. As in the malicious adversary model, corrupted parties may deviate arbitrarily from the protocol specification but the protocol ensures that cheating is detected with a fixed probability, called *deterrence factor* ϵ . The idea of covert security is that adversaries fear to be detected, e.g., due to reputation issues, and thus refrain from cheating.

Although cheating can be detected in covert security, a party of the protocol cannot transfer the knowledge about malicious behavior to other (external) parties. This shortcoming was addressed by Asharov and Orlandi [AO12] with the notion of *covert security with public verifiability* (PVC). Informally, PVC enables honest parties to create a publicly verifiable certificate about the detected malicious behavior. This certificate can subsequently be checked by any other party (often called *judge*), even if this party did not contribute to the protocol execution. The idea behind this notion is to increase the deterrent effect by damaging the reputation of corrupted parties publicly. PVC secure protocols for the two-party case were presented by [AO12, KM15, ZDH19, HKK⁺19]. Recently, Damgård et al. [DOS20] showed a generic compiler from semi-honest to publicly verifiable covert security for the two-party setting and gave an intuition on how to extend their compiler to the multi-party case. Full specifications of generic compilers from semi-honest to publicly verifiable covert security for multi-party protocols were presented by Faust et al. [FHKS21] and Scholl et al. [SSS21].

Although PVC seems to solve the shortcoming of covert security at first glance, in many settings PVC is not sufficient; especially, if only a digital identity of the parties is known, e.g., in the Internet. In such a setting, a real party can create a new identity without suffering from a damaged reputation in the sequel. Hence, malicious behavior needs to be punished in a different way. A promising approach is to use existing cryptocurrencies to directly link cheating detection to financial punishment without involving trusted third parties; in particular, cryptocurrencies that support so-called *smart contracts*, i.e., programs that enable the transfer of assets based on predefined rules. Similar to PVC, where an external judge verifies cheating by checking a certificate of misbehavior, we envision a smart contract that decides whether a party behaved maliciously or not. In this setting, the task of judging is executed over a distributed blockchain network keeping it incorruptible and verifiable at the same time. Since every instruction executed by a smart contract costs fees, it is highly important to keep the amount of computation performed by a contract small. This aspect is not solely important for execution of smart contracts but in all settings where an external judge charges by the size of the task it gets. Due to this constraint, we cannot straightforwardly adapt PVC protocols to work in this setting, since detection of malicious behavior in existing PVC protocols is performed in a naive way that requires the judge to recompute a whole protocol execution.

Related work. While combining MPC with blockchain technologies is an active research area (e.g., [KB14, BK14, ADMM14]) none of these works deal with realizing the judging process of PVC protocols over a blockchain. The only work connecting covert security with financial punishment thus far is by Zhu et al. [ZDH19], which we describe in a bit more detail below. They combine a two-party garbling protocol with an efficient judge that can be realized via a smart contract. Their construction leverages strong security primitives, like a malicious secure oblivious transfer for the transmission of input wires, to ensure that cheating can only occur during the transmission of the garbled circuit and not in any other part of the two-party protocol. By using a binary search over the transmitted circuit, the parties narrow down the computation step under dispute to a single circuit gate. This process requires $O(\log(|C|))$ interactions, where $|C|$ denotes the circuit size, and enables the judge to resolve the dispute by recomputing only a single circuit gate.

While the approach of Zhu et al. [ZDH19] provides an elegant way to reduce the computational complexity of the judge in case cheating is restricted to a single message, it falls short if multiple messages or even a whole protocol execution is under dispute. As a consequence, their construction is limited in scalability and generality, since it is only applicable to two-party garbling protocols, i.e., neither other semi-honest two-party protocols nor more parties are supported.

Generalizing the ideas of [ZDH19] to work for other protocol types and the multi-party case requires us to address several challenges. First, in [ZDH19] the transmitted garbled circuit under dispute is the result of the completely non-interactive garbling process. In contrast, many semi-honest MPC protocols (e.g., [GMW87, BMR90]) consist of several rounds of interactions that need to be all considered during the verification. Interactivity poses the challenge that multiple messages may be under dispute and the computation of messages performed by parties may depend on data received in previous rounds. Hence, verifications of messages need to consider local computations and internal states of the parties that depend on all previous communication rounds. This task is far more complex than verifying a single public message. Second, supporting more than two parties poses the challenge of resolving a dispute about a protocol execution during which parties might not know the messages sent between a subset of other parties. Third, the transmitted garbled circuit in [ZDH19] is independent of the parties' private inputs. Considering protocols where parties provide secret inputs or messages that depend on these inputs, requires a privacy-preserving verification mechanism to protect parties' sensitive data.

1.1 Contribution

Our first contribution is to introduce a new security notion called *financially backed covert security* (FBC). This notion combines a covertly secure protocol with a mechanism to financially punish a corrupted party if cheating was detected. We formalize financial security by adding two properties to covert security, i.e., *financial accountability* and *financial defamation freeness*. Our notion is similar to the one of PVC; in fact, PVC adds reputational punishment

to covert security via *accountability* and *defamation freeness*. In order to lift these properties to the financial context, FBC requires deposits from all parties and allows for an interactive judge. We present two security games to formalize our introduced properties. While the properties are close to accountability and defamation freeness of PVC, our work for the first time explicitly presents formal security games for these security properties, thereby enabling us to rigorously reason about financial properties in PVC protocols. We briefly compare our new notion to the security definition of Zhu et al. [ZDH19], which is called *financially secure computation*. Zhu et al. follow the approach of simulation-based security by presenting an ideal functionality for two parties that extends the ideal functionality of covert security. In contrast, we present a game-based security definition that is not restricted to the two-party case. While simulation-based definitions have the advantage of providing security under composition, proving a protocol secure under their notion requires to create a full simulation proof which is an expensive task. Instead, our game-based notion allows to re-use simulation proofs of all existing covert and PVC protocols, including future constructions, and to focus on proving financial accountability and financial defamation freeness in a standalone way.

We present transformations from different classes of PVC protocols to FBC protocols. While we could base our transformations on covert protocols, FBC protocols require a property called *prevention of detection dependent abort*, which is not always guaranteed by a covert protocol. The property ensures that a corrupted party cannot abort after learning that her cheating will be detected without leaving publicly verifiable evidence. PVC protocols always satisfy prevention of detection dependent abort. So, by basing our transformation on PVC protocols, we inherit this property.

While the mechanism utilized by [ZDH19] to validate misbehavior is highly efficient, it has only been used for non-interactive algorithms so far, i.e. to validate correctness of the garbling process. We face the challenge of extending this mechanism over an interactive protocol execution while still allowing for efficient dispute resolution such that the judge can be realized via a smart contract. In order to tackle these challenges, we present a novel technique that enables efficient validation of arbitrary complex and interactive protocols given the randomness and inputs of all parties. What's more, we can allow for private inputs if a public transcript of all protocol messages is available. We utilize only standard cryptographic primitives, in particular, commitments and signatures.

We differentiate existing PVC protocols according to whether the parties provide private inputs or not. The former protocols are called *input-dependent* and the latter ones *input-independent*. Input-independent protocols are typically used to generate correlated randomness. Further, all existing PVC protocols incorporate some form of common public transcript. Input-dependent protocols require a common public transcript of messages. In contrast, for input-independent protocols, it is enough to agree on the hashes of all sent messages. While it is not clear, if it is possible to construct PVC protocols without any form of public transcript, we construct FBC protocols providing this property. We achieve this

by exploiting the interactivity of the judge, which is non-interactive in PVC. Based on the above observations, we define the following three classes of FBC protocols, for which we present transformations from PVC protocols.

Class 1: The first class contains *input-independent* protocols during which parties learn hashes of all protocol messages such that they agree on a common *transcript of message hashes*.

Class 2: The second class contains *input-dependent* protocols with a public *transcript of messages*. In contrast to class 1, parties may provide secret inputs and share a common view on all messages instead of a common view on hashes only.

Class 3: The third class contains input-independent protocols where parties do not learn any information about messages exchanged between a subset of other parties (cf. class 1). As there are no PVC protocol fitting into this class, we first convert PVC protocols matching the requirements of class 1 into protocols without public transcripts and second leverage an interactive punishment procedure to transform the resulting protocols into FBC protocols without public transcripts. Our FBC protocols benefit from this property since parties have to send all messages only to the receiver and not to all other parties. This effectively reduces the concrete communication complexity by a factor depending on the number of parties. In the optimistic case, if there is no cheating, we get this benefit without any overhead in the round complexity.

For each of our constructions, we provide a formal specification and a rigorous security analysis; the ones of the second class can be found in the full version of this paper. This is in contrast to the work of [ZDH19] which lacks a formal security analysis for financially secure computation. We stress that all existing PVC multi-party protocols can be categorized into class 1 and 2. Additionally, by combining any of the transformations from [DOS20, FHKS21, SSS21], which compile semi-honest protocols into PVC protocols, our constructions can be used to transform these protocol into FBC protocols.

The resulting FBC protocols for class 1 and 2 allow parties to non-interactively send evidence about malicious behavior to the judge. As the judge entity in these two classes is non-interactive, techniques from our transformations are of independent interest to make PVC protocols more efficient. Since, in contrast to class 1 and 2, there is no public transcript present in protocols of class 3, we design an interactive process involving the judge entity to generate evidence about malicious behavior. For all protocols, once the evidence is interactively or non-interactively created, the judge can efficiently resolve the dispute by recomputing only a single protocol message regardless of the overall computation size. We can further reduce the amount of validation to a single program instruction, e.g., a gate in a circuit, by prepending an interactive search procedure. This extension is presented in the full version of this paper.

Finally, we provide a smart contract implementation of the judging party in Ethereum and evaluate its gas costs (cf. Section 8). The evaluation shows the

practicability, e.g., in the three party setting, with optimistic execution costs of 533 k gas. Moreover, we show that the dispute resolution of our solution is highly scalable in regard to the number of parties, the number of protocol rounds and the protocol complexity.

1.2 Technical Overview

In this section, we outline the main techniques used in our work and present the high-level ideas incorporated into our constructions. We start with an overview of the new notion of *financially backed covert security*. Then, we present a first attempt of a construction over a blockchain and outline the major challenges. Next, we describe the main techniques used in our constructions for PVC protocols of classes 1 and 2 and finally elaborate on the bisection procedure required for the more challenging class 3.

Financially backed covert security. We recall that, a publicly verifiable covertly secure (PVC) protocol $(\pi_{\text{cov}}, \text{Blame}, \text{Judge})$ consists of a covertly secure protocol π_{cov} , a blaming algorithm **Blame** and a judging algorithm **Judge**. The blaming algorithm produces a certificate *cert* in case cheating was detected and the judging algorithm, upon receiving a valid certificate, outputs the identity of the corrupted party. The algorithm **Judge** of a PVC protocol is explicitly defined as non-interactive. Therefore, *cert* can be transferred at any point in time to any third party that executes **Judge** and can be convinced about malicious behavior if the algorithm outputs the identity of a corrupted party.

In contrast to PVC, *financially backed covert security* (FBC) works in a model where parties own assets which can be transferred to other parties. This is modelled via a ledger entity \mathcal{L} . Moreover, the model contains a trusted judging party \mathcal{J} which receives deposits before the start of the protocol and adjudicates in case of detected cheating. We emphasize that the entity \mathcal{J} , which is a single trusted entity interacting with all parties, is not the same as the algorithm **Judge** of a PVC protocol, which can be executed non-interactively by any party. An FBC protocol $(\pi'_{\text{cov}}, \text{Blame}', \text{Punish})$ consists of a covertly secure protocol π'_{cov} , a blaming algorithm **Blame'** and an interactive punishment protocol **Punish**. Similar to PVC, the blaming algorithm **Blame'** produces a certificate *cert'* that is used as an input to the interactive punishment protocol. **Punish** is executed between the parties and the judge \mathcal{J} . If all parties behave honestly during the execution of π'_{cov} , \mathcal{J} sends the deposited coins back to all parties after the execution of **Punish**. In case cheating is detected during π'_{cov} , the judge \mathcal{J} burns the coins of the cheating party.

First attempt of an instantiation over a blockchain. Blockchain technologies provide a convenient way of handling monetary assets. In particular, in combination with the execution of smart contracts, e.g., offered by Ethereum [W⁺14], we envision to realize the judging party \mathcal{J} as a smart contract. A first attempt of designing the punishment protocol is to implement \mathcal{J} in a way, that the judge just gets the certificate generated by the PVC protocol's blame algorithm and

executes the PVC protocol’s *Judge*-algorithm. However, the *Judge*-algorithm of all existing PVC protocols recomputes a whole protocol instance and compares the output with a common transcript on which all parties agree beforehand. As computation of a smart contract costs money in form of transaction fees, recomputing a whole protocol is prohibitively expensive. Therefore, instead of recomputing the whole protocol, we aim for a punishment protocol that facilitates a judging party \mathcal{J} which needs to recompute just a single protocol step or even a single program instruction, e.g., a gate in a circuit. The resulting judge becomes efficient in a way that it can be practically realized via a smart contract.

FBC protocols with efficient judging from PVC protocols. In this work, we present three transformations from PVC protocols to FBC protocols. Our transformations start with PVC protocols providing different properties which we use to categorize these protocols into three classes. We model the protocol execution in a way such that every party’s behavior is deterministically defined by her input, her randomness and incoming messages. More precisely, we define the initial state of a party as her input and some randomness and compute the next state according to the state of the previous round and the incoming messages of the current round. Our first two transformations build on PVC protocols where the parties share a public transcript of the exchanged messages resp. message hashes. Additionally, parties send signed commitments on their intermediate states to all parties. The opening procedure ensures that correctly created commitments can be opened – falsely created commitments open to an invalid state that is interpreted as an invalid message. By sending the internal state of some party P_m for a single round together with the messages received by P_m in the same round to the judging party, the latter can efficiently verify malicious behavior by recomputing just a single protocol step. The resulting punishment protocol is efficient and can be executed without contribution of the cheating party.

Interactive punishment protocol to support private transcripts. Our third transformation compiles input-independent PVC protocols with a public transcript into protocols where no public transcript is known to the parties. The lack of a public transcript makes the punishment protocol more complicated. Intuitively, since an honest party has no signed information about the message transcript, she cannot provide verifiable data about the incoming message used to calculate a protocol step. Therefore, we use the technique of an interactive bisection protocol which was first used in the context of verifiable computing by Canetti et al. [CRR11] and subsequently by many further constructions [KGC⁺18, TR19, ZDH19, EFS20]. While the bisection technique is very efficient to narrow down disagreement, it was only used for non-interactive algorithms so far. Hence, we extend this technique to support also interactive protocols. In particular, in our work, we use a bisection protocol to allow two parties to efficiently agree on a common message history. To this end, both parties, the accusing and the accused one, create a Merkle tree of their emulated message history up to the disputed message and submit the corresponding root. If they agree on the message history, the accusation can be validated by ref-

erence to this history. If they disagree, they perform a bisection search over the proposed history that determines the first message in the message history, they disagree on, while automatically ensuring that they agree on all previous messages. Hence, the judge can verify the message that the parties disagree on based on the previous messages they agree on. At the end of both interactions, the judge can efficiently resolve the dispute by recomputing just a single step.

2 Preliminaries

We start by introducing notation and cryptographic primitives used in our construction. Moreover, we provide the definition of covert security and publicly verifiable covert security in the full version of this paper.

We denote the computational security parameter by κ . Let n be some integer, then $[n] = \{1, \dots, n\}$. Let $i \in [n]$, then we use the notation $j \neq i$ for $j \in [n] \setminus \{i\}$. A function $\text{negl}(n) : \mathbb{N} \rightarrow \mathbb{R}$ is *negligible* in n if for every positive integer c there exists an integer n_0 such that $\forall n > n_0$ it holds that $\text{negl}(n) < \frac{1}{n^c}$. We use the notation $\text{negl}(n)$ to denote a negligible function.

We define $\text{REAL}_{\pi, \mathcal{A}(z), \mathcal{I}}(\bar{x}, 1^\kappa)$ to be the output of the execution of an n -party protocol π executed between parties $\{P_i\}_{i \in [n]}$ on input $\bar{x} = \{x_i\}_{i \in [n]}$ and security parameter κ , where \mathcal{A} on auxiliary input z corrupts parties $\mathcal{I} \subset \{P_i\}_{i \in [n]}$. We further specify $\text{OUTPUT}_j(\text{REAL}_{\pi, \mathcal{A}(z), \mathcal{I}}(\bar{x}, 1^\kappa))$ to be the output of party P_j for $j \in [n]$.

Our protocol utilizes a signature scheme (**Generate**, **Sign**, **Verify**) that is *existentially unforgeable under chosen-message attacks*. We assume that each party executes the **Generate**-algorithm to obtain a key pair (pk, sk) before the protocol execution. Further, we assume that all public keys are published and known to all parties while the secret keys are kept private. To simplify the protocol description we denote signed messages with $\langle x \rangle_i$ instead of $(x, \sigma := \text{Sign}_{\text{sk}_i}(x))$. The verification is therefore written as $\text{Verify}(\langle x \rangle_i)$ instead of $\text{Verify}_{\text{pk}_i}(x, \sigma)$. Further, we make use of a hash function $H(\cdot) : \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$ that is collision resistant.

We assume a synchronous communication model, where communication happens in rounds and all parties are aware of the current round. Messages that are sent in some round k arrive at the receiver in round $k + 1$. Since we consider a rushing adversary, the adversary learns the messages sent by honest parties in round k in the same round and hence can adapt her own messages accordingly. We denote a message sent from party P_i to party P_j in round k of some protocol instance denoted with ℓ as $\text{msg}_{(\ell, k)}^{(i, j)}$. The hash of this message is denoted with $\text{hash}_{(\ell, k)}^{(i, j)} := H(\text{msg}_{(\ell, k)}^{(i, j)})$.

A *Merkle tree* over an ordered set of elements $\{x_i\}_{i \in [N]}$ is a labeled binary hash tree, where the i -th leaf is labeled by x_i . We assume N to be an integer power of two. In case the number of elements is not a power of two, the set can be padded until N is a power of two. For construction of Merkle trees, we make use of the collision-resistant hash function $H(\cdot) : \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$.

Formally, we define a Merkle tree as a tuple of algorithms (MTree , MRoot , MProof , MVerify). Algorithm MTree takes as input a computational security parameter κ as well as a set of elements $\{x_i\}_{i \in [N]}$ and creates a Merkle tree mTree . To ease the notation, we will omit the security parameter and implicitly assume it to be provided. Algorithm MRoot takes as input a Merkle tree mTree and returns the root element root of tree mTree . Algorithm MProof takes as input a leaf x_j and Merkle tree mTree and creates a Merkle proof σ showing that x_j is the j -th leaf in mTree . Algorithm MVerify takes as input a proof σ , an index i , a root root and a leaf x^* and returns true iff x^* is the i -th leaf of a Merkle tree with root root .

A Merkle Tree satisfies the following two requirements. First, for each Merkle tree mTree created over an arbitrary set of elements $\{x_i\}_{i \in [N]}$, it holds that for each $j \in [N]$ $\text{MVerify}(\text{MProof}(x_j, \text{mTree}), j, \text{MRoot}(\text{mTree}), x_j) = \text{true}$. We call this property *correctness*. Second, for each Merkle tree mTree with root $\text{root} := \text{MRoot}(\text{mTree})$ created over an arbitrary set of elements $\{x_i\}_{i \in [N]}$ with security parameter κ it holds that for each polynomial time algorithm adversary \mathcal{A} outputting an index j^* , leaf $x^* \neq x_{j^*}$ and proof σ^* the probability that $\text{MVerify}(\sigma^*, j^*, \text{MRoot}(\text{mTree}), x^*) = \text{true}$ is $\text{negl}(\kappa)$. We call this property *binding*.

3 Financially Backed Covert Security

In the following, we specify the new notion of *financially backed covert security*. This notion extends covert security by a mechanism of financial punishment. More precisely, once an honest party detects cheating of the adversary during the execution of the covertly secure protocol, there is some corrupted party that is financial punished afterwards. The financial punishment is realized by an interactive protocol Punish that is executed directly after the covertly secure protocol. In order to deal with monetary assets, financially backed covertly secure protocols depend on a public ledger \mathcal{L} and a trusted judge \mathcal{J} . The former can be realized by distributed ledger technologies, such as blockchains, and the latter by a smart contract executed on the said ledger. In the following, we describe the role of the ledger and the judging party, formally define financially backed covert security and outline techniques to prove financially backed covert security.

3.1 The Ledger and Judge

An inherent property of our model is the handling of assets and asset transfers based on predefined conditions. Nowadays, distributed ledger technologies like blockchains provide convenient means to realize this functionality. We model the handling of assets resp. coins via a ledger entity denoted by \mathcal{L} . The entity stores a balance of coins for each party and transfers coins between parties upon request. More precisely, \mathcal{L} stores a balance $b_i^{(t)}$ for each party P_i at time t . For the security definition presented in Section 3.2, we are in particular interested in the balances before the execution of the protocol π , i.e., $b_i^{(\text{pre})}$, and after the

execution of the protocol **Punish**, i.e., $b_i^{(\text{post})}$. The balances are public such that every party can query the amount of coins for any party at the current time. In order to send coins to another party, a party interacts with \mathcal{L} to trigger the transfer.

While we consider the ledger as a pure storage of balances, we realize the conditional transfer of coins based on some predefined rules specified by the protocol **Punish** via a judge \mathcal{J} . In particular, \mathcal{J} constitutes a trusted third party that interacts with the parties of the covertly secure protocol. More precisely, we require that each party sends some fixed amount of coins as deposit to \mathcal{J} before the covertly secure protocol starts. During the covertly secure protocol execution, the judge keeps the deposited coins but does not need to be part of any interaction. After the execution of the covertly secure protocol, the judge plays an important role in the punishment protocol **Punish**. In case any party detects cheating during the execution of the covertly secure protocol, \mathcal{J} acts as an adjudicator. If there is verifiable evidence about malicious behavior of some party, the judge financially punishes the corrupted party by withholding her deposit. Eventually, \mathcal{J} will reimburse all parties with their deposits except those parties that have been proven to be malicious. The rules according to which parties are considered malicious and hence according to which the coins are reimbursed or withheld need to be specified by the protocol **Punish**.

Finally, we emphasize that both entities the ledger \mathcal{L} and the judge \mathcal{J} are considered trusted. This means, the correct functionality of these entities cannot be distorted by the adversary.

3.2 Formal Definition

We work in a model in which a ledger \mathcal{L} and a judge \mathcal{J} as explained above exist. Let π' be an n -party protocol that is covertly secure with deterrence factor ϵ . Let the number of corrupted parties that is tolerated by π' be $m < n$ and the set of corrupted parties be denoted by \mathcal{I} . We define π as an extension of π' , in which all involved parties transfer a fixed amount of coins, d , to \mathcal{J} before executing π' . Additionally, after the execution of π' , all parties execute algorithm **Blame** which on input the view of the honest party outputs a certificate and broadcasts the generated certificate – still as part of π . The certificate is used for both proving malicious behavior, if detected, and defending against being accused for malicious behavior.

After the execution of π , all parties participate in the protocol **Punish**. In case honest parties detected misbehavior, they prove said misbehavior to \mathcal{J} such that \mathcal{J} can punish the malicious party. In case a malicious party blames an honest one, the honest parties participate to prove their correct behavior. Either way, even if there is no blame at all, all honest parties wait to receive their deposits back, which are reimbursed by \mathcal{J} at the end of the punishment protocol **Punish**.

Definition 1 (Financially backed covert security). *We call a triple $(\pi, \text{Blame}, \text{Punish})$ an n -party financially backed covertly secure protocol with*

deterrence factor ϵ computing some function f in the \mathcal{L} and \mathcal{J} model, if the following security properties are satisfied:

1. **Simulatability with ϵ -deterrent:** The protocol π (as described above) is secure against a covert adversary according to the strong explicit cheat formulation with ϵ -deterrent and non-halting detection accurate.
2. **Financial Accountability:** For every PPT adversary \mathcal{A} corrupting parties P_i for $i \in \mathcal{I} \subset [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0, 1\})^{n+1}$ the following holds:
If for any honest party $P_h \in [n] \setminus \mathcal{I}$ it holds that $\text{OUTPUT}_h(\text{REAL}_{\pi, \mathcal{A}(z), \mathcal{I}}(\bar{x}, 1^\kappa)) = \text{corrupted}_*$ ³, then $\exists m \in \mathcal{I}$ such that:

$$\Pr[b_m^{(\text{post})} = b_m^{(\text{pre})} - d] > 1 - \mu(\kappa),$$

where d denotes the amount of deposited coins per party.

3. **Financial Defamation Freeness:** For every PPT adversary \mathcal{A} corrupting parties P_i for $i \in \mathcal{I} \subset [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\bar{x}, z) \in (\{0, 1\})^{n+1}$ and all $j \in [n] \setminus \mathcal{I}$ the following holds:

$$\Pr[b_j^{(\text{post})} < b_j^{(\text{pre})}] < \mu(\kappa).$$

Remark: For simplicity, we assume that the adversary does not transfer coins after sending the deposit to \mathcal{J} . This assumption can be circumvented by restating financial accountability such that the sum of the balances of all corrupted parties (not just the ones involved in the protocol) is reduced by d .

3.3 Proving Security of Financially Backed Covert Security

Our notion of financially backed covert security (FBC) consists of three properties. The simulatability property requires the protocol π , which augments the covertly secure protocol π' , to be covertly secure as well. This does not automatically follow from the security of π' , in particular since π includes the broadcast of certificates in case of detected cheating. Showing simulatability of π guarantees that the adversary does not learn sensitive information from the certificates. Showing that a protocol π satisfies the simulatability property is proven via a simulation proof. In contrast, we follow a game-based approach to formally prove financial accountability and financial defamation freeness. To this end, we introduce two novel security games, Exp^{FA} and Exp^{DF} , in the following. Although these two properties are similar to the accountability and defamation freeness properties of PVC, we are the first to introduce formal security games for any of these properties. While we focus on financial accountability and financial defamation freeness, we note that our approach and our security games can be adapted to suit for the security properties of PVC as well.

³ We use the notation corrupted_* to denote that the output of P_h is corrupted_i for some $i \in \mathcal{I}$. We stress that i does not need to be equal to m of the financial accountability property.

Both security games are played between a challenger \mathcal{C} and an adversary \mathcal{A} . We define the games in a way that allows us to abstract away most of the details of π . In particular, we parameterize the games by two inputs, one for the challenger and one for the adversary. The challenger's input contains the certificates $\{\text{cert}_i\}_{i \in [n] \setminus \mathcal{I}}$ of all honest parties generated by the Blame-algorithm after the execution of π while the adversary's input consists of all malicious parties' views $\{\text{view}_i\}_{i \in \mathcal{I}}$. By introducing the certificates as inputs to the game, we can prove financial accountability and financial defamation freeness independent from proving simulatability of protocol π .

Throughout the execution of the security games, the adversary executes one instance of the punishment protocol `Punish` with the challenger that takes over the roles of all honest and trusted parties, i.e., the honest protocol parties P_h for $h \notin \mathcal{I}$, the judge \mathcal{J} , and the ledger \mathcal{L} . To avoid an overly complex challenger description, we define those parties as separated entities that can be addressed by the adversary separately and are all executed by the challenger: $\{P_h\}_{h \in [n] \setminus \mathcal{I}}$, \mathcal{J} , and \mathcal{L} . In case any entity is supposed to act pro-actively and does not only wait to react to malicious behavior, the entity is invoked by the challenger. Communication between said entities is simulated by the challenger. The adversary acts on behalf of the corrupted parties.

Financial accountability game. Intuitively, financial accountability states that whenever any honest party detects cheating, there is some corrupted party that loses her deposit. Therefore, we require that the output of all honest parties was `corruptedm` for $m \in \mathcal{I}$ in the execution of π . If this holds, the security game executes `Punish` as specified by the FBC protocol. Before the execution of `Punish`, the challenger asks the ledger for the balances of all parties and stores them as $\{b_i^{(\text{prePunish})}\}_{i \in [n]}$. Note that `prePunish` denotes the time before `Punish` but after the whole protocol already started. This means, relating to Definition 1, the security deposits are already transferred to \mathcal{J} , i.e., $b_i^{\text{prePunish}} = b_i^{\text{pre}} - d$. After the execution, the challenger \mathcal{C} again reads the balances of all parties storing them as $\{b_i^{(\text{post})}\}_{i \in [n]}$. If $b_m^{(\text{post})} = b_m^{(\text{prePunish})} + d$ for all $m \in \mathcal{I}$, i.e., all corrupted parties get their deposits back, the adversary wins and \mathcal{C} outputs 1, otherwise \mathcal{C} outputs 0. A protocol satisfies the financial accountability property as stated in Definition 1 if for each adversary \mathcal{A} running in time polynomial in κ the probability that \mathcal{A} wins game Exp^{FA} is at most negligible, i.e., if $\Pr[\text{Exp}^{\text{FA}}(\mathcal{A}, \kappa) = 1] \leq \text{negl}(\kappa)$.

Financial defamation freeness game. Intuitively, financial defamation freeness states that an honest party can never lose her deposit as a result of executing the `Punish` protocol. The security game is executed in the same way as the financial accountability game. It only differs in the winning conditions for the adversary. After the execution \mathcal{C} checks the balances of the honest parties. If $b_h^{(\text{post})} < b_h^{(\text{prePunish})} + d$ for at least one $h \in [n] \setminus \mathcal{I}$, the adversary wins and the challenger outputs 1, otherwise \mathcal{C} outputs 0. A protocol satisfies the financial defamation freeness property as stated in Definition 1 if for each adversary \mathcal{A} running in time polynomial in κ the probability that \mathcal{A} wins game Exp^{DF} is at most negligible, i.e. if $\Pr[\text{Exp}^{\text{DF}}(\mathcal{A}, \kappa) = 1] \leq \text{negl}(\kappa)$.

4 Features of PVC Protocols

We present transformations from different classes of *publicly verifiable covertly secure* multi-party protocols (PVC) to *financially backed covertly secure* protocols (FBC). As our transformations make use of concrete features of the PVC protocol (e.g., the exchanged messages), we cannot use the PVC protocol in a block-box way. Instead, we model the PVC protocol in an abstract way, stating features that are required by our constructions. In the remainder of this section, we present these features in detail and describe how we model them. We note that all existing PVC multi-party protocols [DOS20, FHKS21, SSS21] provide the features specified in this section.

4.1 Cut-and-Choose

Although not required per definition of PVC, a fundamental technique used by all existing PVC protocols is the *cut-and-choose* approach that leverages a semi-honest protocol by executing t instances of the semi-honest protocol in parallel. Afterwards, the views (i.e., input and randomness) of the parties is revealed in s instances. This enables parties to detect misbehavior with probability $\epsilon = \frac{s}{t}$. PVC protocols can be split into protocols where parties provide private inputs and those where parties do not have secret data. While cut-and-choose for input-independent protocols, i.e., those where parties do not have private inputs, work as explained on a high level before, the approach must be utilized in such a way that input privacy is guaranteed for input-dependent protocols. However, for both classes of protocols, a cheat detection probability of $\epsilon = \frac{s}{t}$ can be achieved. We elaborate more on the two variants and provide details about them in the full version of this paper.

4.2 Verification of Protocol Executions

An important feature of PVC protocols based on cut-and-choose is to enable parties to verify the execution of the opened protocol instances. This requires parties to emulate the protocol messages and compare them with the messages exchanged during the real execution. In order to emulate honest behavior, we need the protocol to be derandomized.

Derandomization of the protocol execution. In general, the behavior of each party during some protocol execution depends on the party’s private input, its random tape and all incoming messages. In order to enable parties to check the behavior of other parties in retrospect, the actions of all parties need to be made deterministic. To this end, we require the feature of a PVC protocol that all random choices of a party P_i in a protocol instance are derived from some random seed seed_i using a pseudorandom generator (PRG). The seed seed_i is fixed before the beginning of the execution. It follows that the generated outgoing messages are computed deterministically given the seed seed_i , the secret input and all incoming messages.

State evolution. Corresponding to our communication model (cf. Section 2), the internal states of the parties in a semi-honest protocol instance evolve in rounds. For each party P_i , for $i \in [n]$, and each round $k > 0$ the protocol defines a state transition computeRound_k^i that on input the previous internal state $\text{state}_{(k-1)}^{(i)}$ and the set of incoming messages $\{\text{msg}_{(k-1)}^{(j,i)}\}_{j \neq i}$ computes the new internal state $\text{state}_{(k)}^{(i)}$ and the set of outgoing messages $\{\text{msg}_{(k)}^{(i,j)}\}_{j \neq i}$. Based on the derandomization feature, the state transition is deterministic, i.e., all random choices are derived from a random seed included in the internal state of a party. Each party starts with an initial internal state that equals its random seed seed_i and its secret input x_i . In case no secret input is present (i.e., in the input-independent setting) or no message is sent, the value is considered to be a dummy symbol (\perp). We denote the set of all messages sent during a protocol instance by *protocol transcript*. Summarizing, we formally define

$$\begin{aligned} \text{state}_{(0)}^{(i)} &\leftarrow (\text{seed}_i, x_i) \\ \{\text{msg}_{(0)}^{(j,i)}\}_{j \in [n] \setminus \{i\}} &\leftarrow \{\perp\}_{j \in [n] \setminus \{i\}} \\ (\text{state}_{(k)}^{(i)}, \{\text{msg}_{(k)}^{(i,j)}\}_{j \in [n] \setminus \{i\}}) &\leftarrow \text{computeRound}_k^i(\text{state}_{(k-1)}^{(i)}, \{\text{msg}_{(k-1)}^{(j,i)}\}_{j \in [n] \setminus \{i\}}). \end{aligned}$$

Protocol emulation. In order to check for malicious behavior, parties locally emulate the protocol execution of the opened instances and compare the set of computed messages with the received ones. In case some involved parties are not checked (e.g., in the input-dependent setting), the emulation gets their messages as input and assumes them to be correct. In this case, in order to ensure that each party can run the emulation, it is necessary that each party has access to all messages sent in the opened instance (cf. Section 4.4).

To formalize the protocol emulation, we define for each n -party protocol π with R rounds two emulation algorithms. The first algorithm $\text{emulate}_\pi^{\text{full}}$ emulates all parties while the second algorithm $\text{emulate}_\pi^{\text{part}}$ emulates only a partial subset of the parties and considers the messages of all other parties as correct. We formally define them as

$$\begin{aligned} (\{\text{msg}_{(k)}^{(i,j)}\}_{k,i,j \neq i}, \{\text{state}_{(k)}^{(i)}\}_{k,i}) &\leftarrow \text{emulate}_\pi^{\text{full}}(\{\text{state}_{(0)}^{(i)}\}_i) \quad \text{and} \\ (\{\text{msg}_{(k)}^{(i,j)}\}_{k,i,j \neq i}, \{\text{state}_{(k)}^{(\hat{i})}\}_{k,\hat{i}}) &\leftarrow \text{emulate}_\pi^{\text{part}}(O, \{\text{state}_{(0)}^{(\hat{i})}\}_{\hat{i}}, \{\text{msg}_{(k)}^{(i^*,j)}\}_{k,i^*,j \neq i^*}) \end{aligned}$$

where $k \in [R]$, $i, j \in [n]$, $\hat{i} \in O$ and $i^* \in [n] \setminus O$. O denotes the set of opened parties.

4.3 Deriving the Initial States

As a third feature, we require a mechanism for the parties of a PVC protocol to learn the initial states of all opened parties in order to perform the protocol emulation (cf. Section 4.2). Since PVC prevents detection dependent abort, parties learn the initial state even if the adversary aborts after having learned the cut-and-choose selection. Existing multi-party PVC protocols provide this feature

by either making use of oblivious transfer or time-lock puzzles as in [DOS20] resp. [FHKS21, SSS21]. We elaborate on these protocols in the full version of this paper.

To model this behavior formally, we define the abstract tuples $\text{initData}^{\text{core}}$ and $\text{initData}^{\text{aux}}$ as well as the algorithm derivelnit . $\text{initData}_{(i)}^{\text{core}}$ represents data each party holds that should be signed by P_i and can be used to derive the initial state of party P_i in a single protocol instance (e.g., a signed time-lock puzzle). $\text{initData}_{(i)}^{\text{aux}}$ represents the additional data all parties receive during the PVC protocol that can be used to interpret $\text{initData}_{(i)}^{\text{core}}$ (e.g., the verifiable solution of the time-lock puzzle). Finally, derivelnit is an algorithm that on input $\text{initData}_{(i)}^{\text{core}}$ and $\text{initData}_{(i)}^{\text{aux}}$ derives the initial state of party P_i (e.g., verifying the solution of the puzzle). Instead of outputting an initial state, the algorithm derivelnit can also output bad or \perp . The former states that party P_i misbehaved during the PVC protocol by providing inconsistent data. The symbol \perp states that the input to derivelnit has been invalid which can only occur if $\text{initData}_{(i)}^{\text{core}}$ or $\text{initData}_{(i)}^{\text{aux}}$ have been manipulated.

Similar to commitment schemes, our abstraction satisfies a *binding* and *hiding* requirement, i.e., it is computationally *binding* and computationally *hiding*. The binding property requires that the probability of any polynomial time adversary finding a tuple (x, y_1, y_2) such that $\text{derivelnit}(x, y_1) \neq \perp$, $\text{derivelnit}(x, y_2) \neq \perp$, and $\text{derivelnit}(x, y_1) \neq \text{derivelnit}(x, y_2)$ is negligible. The hiding property requires that the probability of a polynomial time adversary finding for a given $\text{initData}^{\text{core}}$ a $\text{initData}^{\text{aux}}$ such that $\text{derivelnit}(\text{initData}^{\text{core}}, \text{initData}^{\text{aux}}) \neq \perp$ is negligible.

4.4 Public Transcript

A final feature required by PVC protocols of class 1 and 2 is the availability of a common public transcript. We define three levels of transcript availability. First, a *common public transcript of messages* ensures that all parties hold a common transcript containing all messages that have been sent during the execution of a protocol instance. Every protocol can be transformed to provide this feature by requiring all parties to send all messages to all other parties and defining a fixed ordering on the sent messages – we consider an ordering of messages by the round they are sent, the index of the sender, and the receiver’s index in this sequence. If messages should be secret, each pair of parties executes a secure key exchange as part of the protocol instance and then encrypts messages with the established keys. Agreement is achieved by broadcasting signatures on the transcript, e.g., via signing the root of a Merkle tree over all message hashes as discussed in [FHKS21] and required in our transformations. Second, a *common public transcript of hashes* ensures that all parties hold a common transcript containing the hashes of all messages sent during the execution of a protocol instance. This feature is achieved similar to the transcript of messages but parties only send message hashes to all parties that are not the intended receiver. Finally, the *private transcript* does not require any agreement on the transcript of a protocol instance.

Currently, all existing multi-party PVC protocols either provide a common public transcript of messages [DOS20, FHKS21] or a common public transcript of hashes [SSS21]. However, [DOS20] and [FHKS21] can be trivially adapted to provide just a common public transcript of hashes.

5 Building Blocks

In this section, we describe the building blocks for our financially backed covertly secure protocols. In the full version of this paper, we show security of the building blocks and that incorporating the building blocks into the PVC protocol does not affect the protocol’s security.

5.1 Internal State Commitments

To realize the judge in an efficient way, we want it to validate just a single protocol step instead of validating a whole instance. Existing PVC protocols prove misbehavior in a naive way by allowing parties to show that some other party P_j had an initial state $\text{state}_{(0)}^{(j)}$. Based on the initial state, the judge recomputes the whole protocol instance. In contrast to this, we incorporate a mechanism that allows parties to prove that P_j has been in state $\text{state}_{(k)}^{(j)}$ in a specific round k where misbehavior was detected. Then, the judge just needs to recompute a single step. To this end, we require that parties commit to each intermediate internal state during the execution of each semi-honest instance in a publicly verifiable way. In particular, in each round k of each semi-honest instance ℓ , each party P_i sends a hash of its internal state to all other parties using a collision-resistant hash function $H(\cdot)$, i.e., $H(\text{state}_{(\ell,k)}^{(i)})$. At the end of a protocol instance each party P_h creates a Merkle tree over all state hashes, i.e., $\text{sTree}_\ell := \text{MTree}(\{\text{hash}_{(\ell,k)}^{(i)}\}_{k \in [R], i \in [n]})$, and broadcasts a signature on the root of this tree, i.e., $\langle \text{MRoot}(\text{sTree}_\ell) \rangle_h$.

5.2 Signature Encoding

Our protocol incorporates signatures in order to provide evidence to the judge \mathcal{J} about the behavior of the parties. Without further countermeasures, an adversary can make use of signed data across multiple instances or rounds, e.g., she could claim that some message msg sent in round k has been sent in round k' using the signature received in round k . To prevent such an attack, we encode signed data by prefixing it with the corresponding indices before being signed. Merkle tree roots are prefixed with the instance index ℓ . Message hashes are prefixed with ℓ , the round index k , the sender index i and the receiver index j . Initial state commitments ($\text{initData}_{(\ell,i)}^{\text{core}}$) are prefixed with ℓ and the index i of the party who’s initial state the commitment refers to. The signature verification algorithm automatically checks for correct prefixing. The indices are derived from the super- and subscripts. If one index is not explicitly provided, e.g., in case only one instance is executed, the index is assumed to be 1.

5.3 Bisection of Trees

Our constructions make heavily use of Merkle trees to represent sets of data. This enables parties to efficiently prove that chunk of data is part of a set by providing a Merkle proof showing that the chunk is a leaf of the corresponding Merkle tree. In case two parties disagree about the data of a Merkle tree which should be identical, we use a bisection protocol Π_{BS} to narrow down the dispute to the first leaf of the tree on which they disagree. This helps a judging party to determine the lying party by just verifying a single data chunk in contrast to checking the whole data. The technique of bisecting was first used by Canetti et al. [CRR11] in the context of verifiable computing. Later, the technique was used in [KGC⁺18, TR19, EFS20].

The protocol is executed between a party P_b with input a tree mTree_b , a party P_m with input a tree mTree_m and a trusted judge \mathcal{J} announcing three public inputs: root^j , the root of mTree_j as claimed by P_j for $j \in \{b, m\}$, and width , the width of the trees, i.e., the number of leaves. The protocol returns the index z of the first leaf at which mTree_b and mTree_m differentiate, the leaf hash_z^m at position z of mTree_m , and the common leaf $\text{hash}_{(z-1)}$ at position $z - 1$. The latter is \perp if $z = 1$. Let $\text{node}(\text{mTree}, x, y)$ be the node of a tree mTree at position x of layer y – positions start with 1. The protocol is executed as follows:

Protocol Bisection Π_{BS}

1. \mathcal{J} initializes layer variable $y := 1$, position variable $x := 1$, last agreed hash $\text{hash}^a := \perp$, and $\text{depth} := \lceil \log_2(\text{width}) \rceil + 1$
2. All parties repeat this step while $y \leq \text{depth}$:
 - (a) Both P_j (for $j \in \{b, m\}$) send $\text{hash}^j := \text{node}(\text{mTree}_j, x, y)$ and $\sigma^j := \text{MProof}(\text{hash}^j, \text{mTree}_j)$ to \mathcal{J} .
 - (b) If $\text{MVerify}(\text{hash}^j, x, \text{root}^j, \sigma^j) = \text{false}$ (for $j \in \{b, m\}$), \mathcal{J} discards the message from P_j .
 - (c) If $y = \text{depth}$, \mathcal{J} keeps hash^b and hash^m and sets $y = y + 1$.
 - (d) If $y < \text{depth}$ and $\text{hash}^b = \text{hash}^m$, \mathcal{J} sets $x = (2 \cdot x) + 1$ and $y = y + 1$.
 - (e) If $y < \text{depth}$ and $\text{hash}^b \neq \text{hash}^m$, \mathcal{J} sets $x = (2 \cdot x) - 1$ and $y = y + 1$.
3. If $\text{hash}^b = \text{hash}^m$
 - \mathcal{J} sets $z := x + 1$ and $\text{hash}_{(z-1)} := \text{hash}^b$.
 - P_m sends $\text{hash}_z^m := \text{node}(\text{mTree}_m, z, \text{depth})$ and $\sigma := \text{MProof}(\text{hash}_z^m, \text{mTree}_m)$ to \mathcal{J} .
 - If $\text{MVerify}(\text{hash}_z^m, z, \text{root}^m, \sigma) = \text{false}$, \mathcal{J} discards. Otherwise \mathcal{J} stores hash_z^m .
4. If $\text{hash}^b \neq \text{hash}^m$
 - \mathcal{J} sets $z := x$ and $\text{hash}_z^m := \text{hash}^m$. If $z = 1$, \mathcal{J} sets $\text{hash}_{(z-1)} := \perp$, and the protocol jumps to step 5.
 - P_m sends $\text{hash}_{(z-1)} := \text{node}(\text{mTree}_m, z - 1, \text{depth})$ and $\sigma := \text{MProof}(\text{hash}_{(z-1)}, \text{mTree}_m)$ to \mathcal{J} .
 - If $\text{MVerify}(\text{hash}_{(z-1)}, z - 1, \text{mTree}_m, \sigma) = \text{false}$, \mathcal{J} discards. Otherwise, \mathcal{J} keeps $\text{hash}_{(z-1)}$.
5. \mathcal{J} announces public outputs z , hash_z^m and $\text{hash}_{(z-1)}$.

6 Class 1: Input-Independent with Public Transcript

Our first transformation builds on input-independent PVC protocols where all parties possess a common public transcript of hashes (cf. Section 4.4) for each checked instance. Since the parties provide no input in these protocols, all parties can be opened. The set of input-independent protocols includes the important class of preprocessing protocols. In order to speed up MPC protocols, a common approach is to split the computation in an *offline* and an *online* phase. During the offline phase, precomputations are carried out to set up some correlated randomness. This phase does not require the actual inputs and can be executed continuously. In contrast, the online phase requires the private inputs of the parties and consumes the correlated randomness generated during the offline phase to speed up the execution. As the online performance is more time critical, the goal is to put as much work as possible into the offline phase. Prominent examples following this approach are the protocols of Damgård et al. [DPSZ12, DKL⁺13] and Wang et al. [WRK17a, WRK17b, YWZ20]. Input-independent PVC protocols with a public transcript can be obtained from semi-honest protocols using the input-independent compilers of Damgård et al. [DOS20] and Faust et al. [FHK21].

In order to apply our construction to an input-independent PVC protocol, π^{PP} , we require π^{PP} to provide some features presented in Section 4 and to have incorporated some of the building blocks described in Section 5. First, we require the PVC protocol to be based on the cut-and-choose approach (cf. Section 4.1). Second, we require the actions of each party P_i in a protocol execution to be deterministically determined by a random seed (cf. Section 4.2). Third, we require that all parties learn the initial states of all other parties in the opened protocol instances (cf. Section 4.3). To this end, the parties receive signed data (e.g., a commitment and decommitment value) to derive the initial states of the other parties. Fourth, parties need to commit to their intermediate internal states during the protocol executions in a publicly verifiable way (cf. Section 5.1). Finally, all signed data match the encoded form specified in Section 5.2.

In order to achieve the public transcript of hashes and the commitments to the intermediate internal states, parties exchange additional data in each round. Formally, whenever some party P_h in round k of protocol instance ℓ transitions to a state $\text{state}_{(\ell,k)}^{(h)}$ with the outgoing messages $\{\text{msg}_{(\ell,k)}^{(h,i)}\}_{i \in [n] \setminus \{h\}}$, then it actually sends the following to P_i :

$$(\text{msg}_{(\ell,k)}^{(h,i)}, \{\text{hash}_{(\ell,k)}^{(h,j)} := H(\text{msg}_{(\ell,k)}^{(h,j)})\}_{j \in [n] \setminus \{h,i\}}, \text{hash}_{(\ell,k)}^{(h)} := H(\text{state}_{(\ell,k)}^{(h)}))$$

Let O denote the set of opened instances. We summarize the aforementioned requirements by specifying the data that the view of any honest party P_h includes. It contains signed data to derive the initial state of all parties for the opened instances (1a), a Merkle tree over the hashes of all messages exchanged within a single instance for all instances (1b), a Merkle tree over the hashes of all intermediate internal states of a single instance for all instances (1c), and signatures from each party over the roots of the message and state trees (1d):

$$\{\langle \text{initData}_{(i,\ell)}^{\text{core}}, \text{initData}_{(i,\ell)}^{\text{aux}} \rangle\}_{\ell \in O, i \in [n]}, \quad (1a)$$

$$\{\text{mTree}_\ell\}_{\ell \in [t]} := \{\text{MTree}(\{\text{hash}_{(\ell,k)}^{(i,j)}\}_{k \in [R], i \in [n], j \neq i})\}_{\ell \in [t]}, \quad (1b)$$

$$\{\text{sTree}_\ell\}_{\ell \in [t]} := \{\text{MTree}(\{\text{hash}_{(\ell,k)}^{(i)}\}_{k \in [R], i \in [n]})\}_{\ell \in [t]} \quad (1c)$$

$$\{\langle \text{MRoot}(\text{mTree}_\ell) \rangle_i\}_{i \in [n], \ell \in [t]} \text{ and } \{\langle \text{MRoot}(\text{sTree}_\ell) \rangle_i\}_{i \in [n], \ell \in [t]}. \quad (1d)$$

We next define the blame algorithm that takes the specified view as input and continue with the description of the punishment protocol afterwards.

The blame algorithm. At the end of protocol π^{PP} , all parties execute the blame algorithm Blame^{PP} to generate a certificate cert . The resulting certificate is broadcasted and the honest party finishes the execution of π^{PP} by outputting cert . The certificate is generated as follows:

Algorithm Blame^{PP}

1. P_h runs $\text{state}_{(\ell,0)}^{(i)} = \text{derivInit}(\text{initData}_{(i,\ell)}^{\text{core}}, \text{initData}_{(i,\ell)}^{\text{aux}})$ for each $i \in [n], \ell \in O$. Let \mathcal{B} be the set of all tuples $(\ell, 0, m, 0)$ such that $\text{state}_{(\ell,0)}^{(m)} = \text{bad}$. If $\mathcal{B} \neq \emptyset$, goto step 4.
2. P_h emulates for each $\ell \in O$ the protocol executions on input the initial states from all parties to obtain the expected messages and the expected intermediate states of all parties, i.e., $(\{\text{msg}_{(\ell,k)}^{(i,j)}\}_{k \in [R], i \in [n], j \neq i}, \{\text{state}_{(\ell,k)}^{(i)}\}_{k,i,j}) := \text{emulate}^{\text{full}}(\{\text{state}_{(\ell,0)}^{(i)}\}_{i \in [n]})$.
3. Let \mathcal{B} be the set of all tuples (ℓ, k, m, i) such that $H(\text{msg}_{(\ell,k)}^{(m,i)}) \neq \text{hash}_{(\ell,k)}^{(m,i)}$ or $H(\text{state}_{(\ell,k)}^{(m)}) \neq \text{hash}_{(\ell,k)}^{(m)}$ – where $\text{hash}_{(\ell,k)}^{(m,i)}$ and $\text{hash}_{(\ell,k)}^{(m)}$ are extracted from mTree_ℓ or sTree_ℓ respectively. In case of an incorrect state hash, set $i = 0$.
4. If $\mathcal{B} = \emptyset$ P_h outputs $\text{cert} := \perp$. Otherwise, P_h picks the tuple (ℓ, k, m, i) from \mathcal{B} with the smallest ℓ, k, m, i in this sequence, sets $k' := k - 1$ and defines variables as follows – variables that are not explicitly defined are set to \perp .

$$\begin{aligned} \text{(Always):} \quad & \text{ids} := (\ell, k, m, i) \\ & \text{initData} := (\langle \text{initData}_{(\ell,m)}^{\text{core}} \rangle_m, \text{initData}_{(\ell,m)}^{\text{aux}}) \\ & \text{root}^{\text{state}} := \langle \text{MRoot}(\text{sTree}_\ell) \rangle_m \\ & \text{root}^{\text{msg}} := \langle \text{MRoot}(\text{mTree}_\ell) \rangle_m \\ \text{(If } k > 0\text{):} \quad & \text{state}_{\text{out}} := (\text{hash}_{(\ell,k)}^{(m)}, \text{MProof}(\text{hash}_{(\ell,k)}^{(m)}, \text{sTree}_\ell)) \\ & \text{msg}_{\text{out}} := (\text{hash}_{(\ell,k)}^{(m,i)}, \text{MProof}(\text{hash}_{(\ell,k)}^{(m,i)}, \text{mTree}_\ell)) \\ \text{(If } k > 1\text{):} \quad & \text{state}_{\text{in}} := (\text{state}_{(\ell,k')}^{(m)}, \text{MProof}(H(\text{state}_{(\ell,k')}^{(m)}), \text{sTree}_\ell)) \\ & \mathcal{M}_{\text{in}} := \{(\text{msg}_{(\ell,k')}^{(j,m)}, \text{MProof}(H(\text{msg}_{(\ell,k')}^{(j,m)}), \text{mTree}_\ell))\}_{j \in [n]} \end{aligned}$$

5. Output $\text{cert} := (\text{ids}, \text{initData}, \text{root}^{\text{state}}, \text{root}^{\text{msg}}, \text{state}_{\text{in}}, \mathcal{M}_{\text{in}}, \text{state}_{\text{out}}, \text{msg}_{\text{out}})$.

The punishment protocol. Each party P_i (for $i \in [n]$) checks if $\text{cert} \neq \perp$. If this is the case, P_i sends cert to \mathcal{J}^{PP} . Otherwise, P_i waits till time T to receive her deposit back. Timeout T is set such that the parties have sufficient time to submit a certificate after the execution of π^{PP} and Blame^{PP} . The judge \mathcal{J}^{PP} is described in the following. The validation algorithms wrongMsg and wrongState and the algorithm getIndex can be found in the full version of this paper. We stress that the validation algorithms wrongMsg and wrongState don't need to recompute a whole protocol execution but only a single step. Therefore, \mathcal{J}^{PP} is very efficient and can, for instance, be realized via a smart contract. To be more precise, the judge is execution without any interaction and runs in computation complexity linear in the protocol complexity. By allowing logarithmic interactions between the judge and the parties, we can further reduce the computation complexity to logarithmic in the protocol complexity. This can be achieved by applying the efficiency improvement described in the full version of this paper.

Judge \mathcal{J}^{PP}

Initialization: The judge has access to public variables n, t, T and the set of parties $\{P_i\}_{i \in [n]}$. Further, it maintains a set cheaters initially set to \emptyset . Prior to the execution of π^{PP} , \mathcal{J}^{PP} has received d coins from each party P_i .

Proof verification: Wait until time T_1 to receive $((\ell, k, m, i), \text{initData}, \langle \text{root}_{(\ell)}^{\text{state}} \rangle_m, \langle \text{root}_{(\ell)}^{\text{msg}} \rangle_m, \text{state}_{in}, \mathcal{M}_{in}, \text{state}_{out}, (\text{hash}, \sigma))$ and do:

1. If $P_m \in \text{cheaters}$, abort.
2. Parse initData to $(\langle \text{initData}_{(\ell, m)}^{\text{core}} \rangle_m, \text{initData}_{(\ell, m)}^{\text{aux}})$ and set $\text{state}_0 = \text{derivelnit}(\text{initData}_{(\ell, m)}^{\text{core}}, \text{initData}_{(\ell, m)}^{\text{aux}})$. If $\text{Verify}(\langle \text{initData}_{(\ell, m)}^{\text{core}} \rangle_m) = \text{false}$ or $\text{state}_0 = \perp$, abort. If $\text{state}_0 = \text{bad}$, add P_m to cheaters and stop.
3. If $\text{Verify}(\langle \text{root}_{(\ell)}^{\text{state}} \rangle_m) = \text{false}$ or $\text{Verify}(\langle \text{root}_{(\ell)}^{\text{msg}} \rangle_m) = \text{false}$, abort.
4. If $i = 0$ and $\text{wrongState}(\text{state}_0, \text{state}_{in}, \text{state}_{out}, \mathcal{M}_{in}, \text{root}_{(\ell)}^{\text{state}}, \text{root}_{(\ell)}^{\text{msg}}, \ell, k, m) = \text{true}$, add P_m to cheaters .
5. If $i > 0$, $\text{MVerify}(\text{hash}, \text{getIndex}(k, m, i), \text{root}_{(\ell)}^{\text{msg}}, \sigma) = \text{true}$ and $\text{wrongMsg}(\text{state}_0, \text{state}_{in}, \text{hash}, \mathcal{M}_{in}, \text{root}_{(\ell)}^{\text{state}}, \text{root}_{(\ell)}^{\text{msg}}, \ell, m, k, i) = \text{true}$, add P_m to cheaters .

Timeout: At time T_1 , send d coins to each party $P_i \notin \text{cheaters}$.

6.1 Security

Theorem 1. *Let $(\pi^{\text{PP}}, \cdot, \cdot)$ be an n -party publicly verifiable covert protocol computing function f with deterrence factor ϵ satisfying the view requirements stated in Eq. (1a)-(1d). Further, let the signature scheme $(\text{Generate}, \text{Sign}, \text{Verify})$ be existentially unforgeable under chosen-message attacks, the Merkle tree satisfies the binding property and the hash function H be collision resistant. Then the protocol π^{PP} together with algorithm Blame^{PP} , protocol $\text{Punish}^{\text{PP}}$ and judge \mathcal{J}^{PP} satisfies financially backed covert security with deterrence factor ϵ according to Definition 1.*

We formally prove Theorem 1 in the full version of this paper.

7 Class 3: Input-Independent with Private Transcript

At the time of writing, there exists no PVC protocol without public transcript that could be directly transformed into an FBC protocol. Moreover, it is not clear, if it is possible to construct a PVC protocol without a public transcript. Instead, we present a transformation from an input-independent PVC protocol with public transcript into an FBC protocol without any form of common public transcript. As in our first transformation, we start with an input-independent PVC protocol π_3^{pvc} that is based on cut-and-choose where parties share a common public transcript. Due to the input-independence, all parties of the checked instances can be opened. However, unlike our first transformation, which utilizes the public transcript, we remove this feature from the PVC protocol as part of the transformation. We denote the protocol that results by removing the public transcript feature from π_3^{pvc} by π_3 . Without having a public transcript, the punishment protocol becomes interactive and more complicated. Intuitively, without a public transcript it is impossible to immediately decide if a message that deviates from the emulation is maliciously generated or is invalid because of a received invalid messages. Note that we still have a common public tree of internal state hashes in our exposition. However, the necessity of this tree can also be removed by applying the techniques presented here that allow us to remove the common transcript.

In order to apply our construction to a protocol π_3 , we require almost the same features of π_3 as demanded in our first transformation (cf. Section 6). For the sake of exposition, we outline the required features here again and point out the differences. First, we require π_3 to be based on the cut-and-choose approach (cf. Section 4.1). Second, we require the actions of each party P_i in a semi-honest instance execution to be deterministically determined by a random seed (cf. Section 4.2). Third, we require that all parties learn the initial states of all other parties in the opened protocol instances (cf. Section 4.3). To this end, the parties receive signed data (e.g., a commitment and decommitment value) to derive the initial states of the other parties. Fourth, parties need to commit to their intermediate internal states during the protocol executions in a publicly verifiable way (cf. Section 5.1). Finally, all signed data match the encoded form specified in Section 5.2.

In contrast to the transformation in Section 6 we no longer require from protocol π_3 that the parties send all messages or message hashes to all other parties. Formally, whenever some party P_h in round k of protocol instance ℓ transitions to a state $\text{state}_{(\ell,k)}^{(h)}$ with the outgoing messages $\{\text{msg}_{(\ell,k)}^{(h,i)}\}_{i \in [n] \setminus \{h\}}$, then it actually sends the following to P_i :

$$(\langle \text{msg}_{(\ell,k)}^{(h,i)} \rangle_h, \text{hash}_{(\ell,k)}^{(h)} := H(\text{state}_{(\ell,k)}^{(h)}))$$

Let O be the set of opened instances. We summarize the aforementioned requirements by specifying the data that the view of any honest party P_h after the execution of π_3 includes. The view contains data to derive the initial state of all parties which is signed by each party for each party and every opened

instance, i.e.,

$$\{(\langle \text{initData}_{(i,\ell)}^{\text{core}} \rangle_j, \text{initData}_{(i,\ell)}^{\text{aux}})\}_{\ell \in O, i \in [n], j \in [n]}, \quad (2a)$$

a Merkle tree over the hashes of all intermediate internal states of a single instance for all instances, i.e.,

$$\{\text{sTree}_\ell\}_{\ell \in [t]} := \{\text{MTree}(\{\text{hash}_{(\ell,k)}^{(i)}\}_{k \in [R], i \in [n]})\}_{\ell \in [t]}, \quad (2b)$$

signatures from each party over the roots of the state trees, i.e.,

$$\{\langle \text{MRoot}(\text{sTree}_\ell) \rangle_i\}_{i \in [n], \ell \in [t]} \quad (2c)$$

and the signed incoming message, i.e.,

$$\mathcal{M} := \{\langle \text{msg}_{(\ell,k)}^{(i,h)} \rangle_i\}_{\ell \in [t], k \in [R], i \in [n] \setminus \{h\}}. \quad (2d)$$

The blame algorithm. At the end of protocol π_3 , all parties first execute an evidence algorithm **Evidence** to generate partial certificates cert' . The partial certificate is a candidate to be used for the punishment protocol and is broadcasted to all other parties as part of π_3 . In case the honest party detects cheating in several occurrences, the party picks the occurrence with the smallest indices (ℓ, k, m, i) (in this sequence). The algorithm to generate partial certificates **Evidence** is formally described as follows:

Algorithm Evidence

1. P_h runs $\text{state}_{(\ell,0)}^{(i)} = \text{derivInit}(\text{initData}_{(i,\ell)}^{\text{core}}, \text{initData}_{(i,\ell)}^{\text{aux}})$ for each $i \in [n], \ell \in O$. Let \mathcal{B} be the set of all tuples $(\ell, 0, m, 0)$ such that $\text{state}_{(\ell,0)}^{(m)} = \text{bad}$. If $\mathcal{B} \neq \emptyset$, goto step 4.
2. P_h emulates for each $\ell \in O$ the protocol executions on input the initial states from all parties to obtain the expected messages and the expected intermediate states of all parties, i.e., $(\{\widetilde{\text{msg}}_{(\ell,k)}^{(i,j)}\}_{k \in [R], i \in [n], j \neq i}, \{\text{state}_{(\ell,k)}^{(i)}\}_{k,i,j}) := \text{emulate}^{\text{full}}(\{\text{state}_{(\ell,0)}^{(i)}\}_{i \in [n]})$.
3. Let \mathcal{B} be the set of all tuples (ℓ, k, m, h) such that $\text{msg}_{(\ell,k)}^{(m,h)} \neq \widetilde{\text{msg}}_{(\ell,k)}^{(m,h)}$ or $H(\text{state}_{(\ell,k)}^{(m)}) \neq \text{hash}_{(\ell,k)}^{(m)}$ – where $\text{msg}_{(\ell,k)}^{(m,h)}$ and $\text{hash}_{(\ell,k)}^{(m)}$ are taken from \mathcal{M} or sTree_ℓ respectively. In case of an invalid state, set $h = 0$.
4. Pick the tuple (ℓ, k, m, i) from \mathcal{B} with the smallest ℓ, k, m, i in this sequence. If $k > 0$ set $\text{msg}_{out} := \langle \text{msg}_{(\ell,k)}^{(m,i)} \rangle_m$. Otherwise, set $\text{msg}_{out} := \perp$.
5. Output partial certificate (ids, msg_{out}) .

Since π_3 does not contain a public transcript of messages, parties can only validate their own incoming message instead of all messages as done in previous approaches. Hence, it can happen that different honest parties generate and broadcast different partial certificates. Therefore, all parties validate the incoming certificates, discard invalid ones and pick the partial certificate cert' with the smallest indices (ℓ, k, m, i) (in this sequence) as their own. If no partial certificate has been received or created, parties set $\text{cert}' := \perp$.

Finally, each honest party executes the blame algorithm Blame^{SP} to create the full certificate that is used for both, blaming a malicious party and defending against incorrect accusations. As in this scenario the punishment protocol requires input of accused honest parties, the blame algorithm returns a certificate even if no malicious behavior has been detected, i.e., if $\text{cert}' = \perp$. The final certificate is generated by appending following data from the view to the certificate: $\{\langle \text{initData}_{(i,\ell)}^{\text{core}} \rangle_j, \text{initData}_{(i,\ell)}^{\text{aux}}\}_{\ell \in \mathcal{O}, i \in [n], j \in [n]}$ (cf. Eq 2a), $\{\text{sTree}_\ell\}_{\ell \in [t]}$ (cf. Eq 2b), and $\{\langle \text{MRoot}(\text{sTree}_\ell) \rangle_i\}_{i \in [n], \ell \in [t]}$ (cf. Eq 2c). All the appended data is public and does not really need to be broadcasted. However, in order to match the formal specification, all parties broadcast their whole certificate. If $\text{cert}' \neq \perp$, the honest party outputs in addition to the certificate corrupted_m .

To ease the specification of the punishment protocol in which parties derive further data from the certificates, we define an additional algorithm mesHistory that uses the messages obtained during the emulation $(\widetilde{\text{msg}})^4$ to compute the message history up to a specific round k' (inclusively) of instance ℓ . We structure the message history in two layers. For each round $k^* < k'$, parties create a Merkle tree of all messages emulated in this round. These trees constitute the bottom layer. On the top layer, parties create a Merkle tree over the roots of the bottom layer trees. This enables parties to agree on all messages of one round making it easier to submit Merkle proofs for messages sent in this round. The message history is composed of the following variables:

$$\begin{aligned} \{\text{mTree}_{k^*}^{\text{round}}\}_{k^* \in [k']} &:= \{\text{MTree}(\{H(\widetilde{\text{msg}}_{(\ell, k^*)}^{(i,j)})\}_{i \in [n], j \neq i})\}_{k^* \in [k']} \\ \text{mTree}_{k'} &:= \text{MTree}(\{\text{MRoot}(\text{mTree}_{k^*}^{\text{round}})\}_{k^* \in [k']}) \\ \text{root}_{k'}^{\text{msg}} &:= \text{MRoot}(\text{mTree}) \end{aligned}$$

Additionally, if $\text{cert}' \neq \perp$, parties compute the following:

$$\begin{aligned} \text{(Always): } \text{initData} &:= (\langle \text{initData}_{(\ell, m)}^{\text{core}} \rangle_m, \text{initData}_{(\ell, m)}^{\text{aux}}) \\ \text{root}^{\text{state}} &:= \langle \text{MRoot}(\text{sTree}_\ell) \rangle_m \\ \text{(If } k > 0\text{): } \text{state}_{\text{out}} &:= (\text{hash}_{(\ell, k)}^{(m)}, \text{MProof}(\text{hash}_{(\ell, k)}^{(m)}, \text{sTree}_\ell)) \\ \text{(If } k > 1\text{): } \text{state}_{\text{in}} &:= (\text{state}_{(\ell, k')}^{(m)}, \text{MProof}(H(\text{state}_{(\ell, k')}^{(m)}), \text{sTree}_\ell)) \\ &(\{\text{mTree}_{k^*}^{\text{round}}\}_{k^* \in [k']}, \text{mTree}_{k'}, \text{root}_{k'}^{\text{msg}}) := \text{mesHistory}(k', \ell) \\ \sigma_{k'} &:= \text{MProof}(\text{MRoot}(\text{mTree}_{k'}^{\text{round}}), \text{mTree}_{k'}) \\ \mathcal{M}_{\text{in}} &:= \{(\widetilde{\text{msg}}_{(\ell, k')}^{(j,m)}, \text{MProof}(H(\widetilde{\text{msg}}_{(\ell, k')}^{(j,m)}), \text{mTree}_{k'}^{\text{round}}))\}_{j \in [n]} \end{aligned}$$

The punishment protocol. The main difficulty of constructing a punishment protocol $\text{Punish}^{\text{SP}}$ for this scenario is that there is no publicly verifiable evidence about messages like a common transcript used in the previous transformations.

⁴ Formally, parties need to re-execute the emulation, as we do not allow them to use any data not included in the certificate.

Hence, incoming messages required for the computation of a particular protocol step cannot be validated directly. Instead, the actions of all parties need to be validated against the emulated actions based on the initial states. This leads to the problem that deviations from the protocol can cause later messages of other honest parties to deviate from the emulated ones as well. Therefore, it is important that the judge disputes the earliest occurrence of misbehavior.

We divide the punishment protocol $\text{Punish}^{\text{SP}}$ into three phases. First, the judge determines the earliest accusation of misbehavior. To this end, if $\text{cert} \neq \perp$ all parties start by sending tuple ids from cert to \mathcal{J}^{SP} and the judge selects the tuple with the smallest indices (ℓ, k, m, i) . This mechanism ensures that either the first malicious message or malicious state hash received by an honest party is disputed or the adversary blames some party at an earlier point. To look ahead, if the adversary blames an honest party at an earlier point, the punishment will not be successful and the malicious blamer will be punished for submitting an invalid accusation. If the adversary blames another malicious party, either one of them will be punished. This mechanism ensures that if an honest party submits an accusation, a malicious party will be punished, even if it is not the honest party's accusation that is disputed.

If there has not been any accusation submitted in the first phase, \mathcal{J}^{SP} reimburses all parties. Otherwise, \mathcal{J}^{SP} defines a blamer P_b , the party that has submitted the earliest accusation, and an accused party P_m . P_b either accuses misbehavior in the initial state, the first round, or in some later round. For the former two, misbehavior can be proven in a straightforward way, similar to our first construction. For the latter, P_b is supposed to submit a proof containing the hash of a tree of the message history up to the disputed round k . P_m can accept or decline the message history depending on whether the tree corresponds to the one emulated by P_m or not. If the tree is accepted, the certificate can be validated as in previous scenarios, with the only difference that incoming messages are validated with respect to the submitted message history tree instead of the common public transcript. In case any party does not respond in time, this party is considered maliciously and is financially punished.

If the message history is declined, the protocol transitions to the third phase. Parties P_b and P_m together with \mathcal{J}^{SP} execute a bisection search in the message history tree to find the first message they disagree on (cf. Section 5.3). By definition they agree on all messages before the disputed one – we call these messages the *agreed sub-tree*. At this step, \mathcal{J}^{SP} can validate the disputed message of the history tree (not the one disputed in the beginning) the same way as done in previous constructions with the only difference that incoming messages are validated with respect to the agreed sub-tree.

The number of interactions is logarithmic while the computation complexity of the judge is linear in the protocol complexity. We can further reduce the computation complexity to be logarithmic in the protocol complexity while still having logarithmic interactions using the efficiency improvements described in the full version of this paper. The judge is defined as follows:

Protocol Punish^{SP}

Phase 1: Determine earliest accusation

1. If $\text{cert} \neq \perp$, P_h sends $\text{ids} := (\ell, k, m, i)$ taken from cert to \mathcal{J}^{SP} which stores (ℓ, k, m, i, h) .
2. \mathcal{J}^{SP} waits till time T to receive message (ℓ, k, m, i) from parties P_b for $b \in [n]$. If no accusations have been received, \mathcal{J}^{SP} sends d coins to each party at time T . Otherwise, \mathcal{J}^{SP} picks the *smallest* tuple (ℓ, k, m, i, b) (ordered in this sequence), sets $k' := k - 1$ and continues with Phase 2.

Timeout: If its P_j 's turn for $j \in \{b, m\}$ and P_j does not respond with a valid message, i.e., one that is not discarded, in time, P_j is considered malicious and \mathcal{J}^{SP} terminates by sending d coins to all parties but P_j .

Phase 2: First evidence

3. If $k < 2$, P_b sends $(\text{initData}, \text{root}^{\text{state}}, \text{state}_{\text{out}}, \langle \text{msg}_{(\ell, k)}^{(m, i)} \rangle_m)$ taken from cert to \mathcal{J}^{SP}
 - (a) \mathcal{J}^{SP} parses initData to $(\langle \text{initData}_{(\ell, m)}^{\text{core}} \rangle_m, \text{initData}_{(\ell, m)}^{\text{aux}})$ and sets $\text{state}_0 = \text{derivelnit}(\text{initData}_{(\ell, m)}^{\text{core}}, \text{initData}_{(\ell, m)}^{\text{aux}})$. If $\text{Verify}(\langle \text{initData}_{(\ell, m)}^{\text{core}} \rangle_m) = \text{false}$ or $\text{state}_0 = \perp$, \mathcal{J}^{SP} discards. If $\text{state}_0 = \text{bad}$, \mathcal{J}^{SP} terminates by sending d coins to all parties but P_m .
 - (b) If $\text{Verify}(\langle \text{root}_{(\ell)}^{\text{state}} \rangle_m) = \text{false}$, \mathcal{J}^{SP} discards.
 - (c) If $i = 0$ and $\text{wrongState}(\text{state}_0, \perp, \text{state}_{\text{out}}, \emptyset, \text{root}_{(\ell)}^{\text{state}}, \perp, \ell, k, m) = \text{false}$, \mathcal{J}^{SP} discards.
 - (d) If $i > 0$, $\text{Verify}(\langle \text{msg}_{(\ell, k)}^{(m, i)} \rangle_m) = \text{false}$ or $\text{wrongMsg}(\text{state}_0, \perp, H(\text{msg}_{(\ell, k)}^{(m, i)}), \emptyset, \text{root}_{(\ell)}^{\text{state}}, \perp, \ell, m, k, i) = \text{false}$, \mathcal{J}^{SP} discards.
 - (e) \mathcal{J}^{SP} terminates by sending d coins to all parties but P_m .
4. Otherwise, P_b sends $(\text{root}^{\text{state}}, \text{state}_{\text{in}}, \text{state}_{\text{out}}, \langle \text{root}_{(\ell)}^{\text{state}} \rangle_m, \text{root}^{\text{msg}}, \text{root}_{k'}^{\text{round}}, \sigma_{k'}, \mathcal{M}_{\text{in}}, \text{msg}_{\text{out}})$ taken from cert to \mathcal{J}^{SP} .
 - (a) P_m executes $\text{mesHistory}(k - 1, \ell)$. Let $\widetilde{\text{root}}^{\text{msg}}$ be the root of the emulated message history tree. If $\text{root}^{\text{msg}} \neq \widetilde{\text{root}}^{\text{msg}}$ P_m sends $\widetilde{\text{root}}^{\text{msg}}$ to \mathcal{J}^{SP} . Otherwise, P_m sends (\perp) .
 - (b) If $\widetilde{\text{root}}^{\text{msg}}$ received by P_m does not equal \perp , \mathcal{J}^{SP} jumps to phase 3.
 - (c) \mathcal{J}^{SP} checks that $\text{Verify}(\langle \text{root}_{(\ell)}^{\text{state}} \rangle_m) = \text{true}$ and $\text{MVerify}(\text{root}_{k'}^{\text{round}}, k', \text{root}^{\text{msg}}, \sigma_{k'}) = \text{true}$ and discards otherwise.
 - (d) If $i = 0$ and $\text{wrongState}(\perp, \text{state}_{\text{in}}, \text{state}_{\text{out}}, \mathcal{M}_{\text{in}}, \text{root}_{(\ell)}^{\text{state}}, \text{root}_{k'}^{\text{round}}, \ell, k, m) = \text{false}$, \mathcal{J}^{SP} discards.
 - (e) If $i > 0$, $\text{Verify}(\langle \text{msg}_{(\ell, k)}^{(m, i)} \rangle_m) = \text{false}$ or $\text{wrongMsg}(\text{state}_0, \text{state}_{\text{in}}, H(\text{msg}_{(\ell, k)}^{(m, i)}), \mathcal{M}_{\text{in}}, \text{root}_{(\ell)}^{\text{state}}, \text{root}_{k'}^{\text{round}}, \ell, m, k, i) = \text{false}$, \mathcal{J}^{SP} discards.
 - (f) \mathcal{J}^{SP} terminates by sending d coins to all parties but P_m .

Phase 3: Dispute the message tree

5. Parties P_b , P_m and \mathcal{J}^{SP} run bisection sub-protocol Π_{BS} on the top-level tree. P_b 's input is the tree with root root^{msg} ; P_m 's the one with root $\widetilde{\text{root}}^{\text{msg}}$. \mathcal{J}^{SP} announces public inputs root^{msg} and width of root^{msg} , $\text{width} := k'$. The output is the first round they disagree on k_2 , the agreed hash $\text{root}_{k_2'}^{\text{round}}$ of leaf with index $k_2' := k_2 - 1$ and the hash $\text{root}_{(b, k_2)}^{\text{round}}$ of leaf with index k_2 as claimed by P_m .

6. Parties P_m , P_b and \mathcal{J}^{SP} run bisection sub-protocol Π_{BS} on the low-level tree. Both, P_m and P_b take as input $\text{mTree}_{k_2}^{\text{round}}$ from their certificate. \mathcal{J}^{SP} announces public inputs $\text{root}_{(b,k_2)}^{\text{round}}$ and the width of the low level tree $\text{width}'n \times (n-1)$. The output is the index x of the first message they disagree on and the hash of this message hash_x as claimed by P_m . The index of the sender of the disputed message is $m_2 := \lceil \frac{x}{n-1} \rceil$ and the index of the receiver $i_2 = x \bmod (n-1)$ if $m_2 > (x \bmod (n-1))$ and $i_2 := (x \bmod (n-1)) + 1$ otherwise.
7. Party P_b define variables as follows – variables that are not explicitly defined are set to \perp .

(Always): $\text{initData}^2 := (\langle \text{initData}_{(\ell, m_2)}^{\text{core}} \rangle_m, \text{initData}_{(\ell, m_2)}^{\text{aux}})$

$\text{root}^{\text{state}} := \langle \text{MRoot}(\text{sTree}_\ell) \rangle_m$

(If $k_2 > 1$): $\text{state}_{in}^2 := (\text{state}_{(\ell, k_2)}^{(m_2)}, \text{MProof}(H(\text{state}_{(\ell, k_2)}^{(m_2)}), \text{sTree}_\ell))$

$\mathcal{M}_{in}^2 := \{(\text{msg}_{(\ell, k_2)}^{(j, m_2)}, \text{MProof}(H(\text{msg}_{(\ell, k_2)}^{(j, m_2)}), \text{mTree}_{k_2}^{\text{round}}))\}_{j \in [n]}$

and sends $(\text{initData}^2, \langle \text{MRoot}(\text{sTree}_\ell) \rangle_m, \text{state}_{in}^2, \mathcal{M}_{in}^2)$ to \mathcal{J}^{SP} .

8. \mathcal{J}^{SP} parses initData^2 to $(\langle \text{initData}_{(\ell, m_2)}^{\text{core}} \rangle_m, \text{initData}_{(\ell, m_2)}^{\text{aux}})$ and sets $\text{state}_{(0)}^{(m_2)} := \text{derivInit}(\text{initData}_{(\ell, m_2)}^{\text{core}}, \text{initData}_{(\ell, m_2)}^{\text{aux}})$. If $\text{Verify}(\langle \text{root}_{(\ell)}^{\text{state}} \rangle_m) = \text{false}$, $\text{Verify}(\langle \text{initData}_{(\ell, m_2)}^{\text{core}} \rangle_m) = \text{false}$ or $\text{state}_{(0)}^{(m_2)} \in \{\perp, \text{bad}\}$, \mathcal{J}^{SP} discards.
9. If $\text{wrongMsg}(\text{state}_{(0)}^{(m_2)}, \text{state}_{in}^2, \text{hash}_x, \mathcal{M}_{in}^2, \text{root}_{(\ell)}^{\text{state}}, \text{root}_{k_2}^{\text{round}}, \ell, m_2, k_2, i_2) = \text{false}$, \mathcal{J}^{SP} discards.
10. \mathcal{J}^{SP} terminates by sending d coins to all parties but P_m .

7.1 Security

Theorem 2. *Let $(\pi_3^{\text{pvc}}, \text{Blame}^{\text{pvc}}, \text{Judge}^{\text{pvc}})$ be an n -party publicly verifiable covert protocol computing function f with deterrence factor ϵ satisfying the view requirements stated in Eq. (2). Further, π_3^{pvc} generates a common public transcript of hashes that is only used for $\text{Blame}^{\text{pvc}}$ and $\text{Judge}^{\text{pvc}}$. Let π_3 be a protocol that is equal to π_3^{pvc} but does not generate a common transcript and instead of calling $\text{Blame}^{\text{pvc}}$ executes the blame procedure explained above (including execution of Evidence and Punish^{SP}). Further, let the signature scheme (Generate, Sign, Verify) be existentially unforgeable under chosen-message attacks, the Merkle tree satisfies the binding property, the hash function H be collision resistant and the bisection protocol Π_{BS} be correct. Then, the protocol π_3 , together with algorithm Blame^{SP} , protocol $\text{Punish}^{\text{SP}}$ and judge \mathcal{J}^{SP} satisfies financially backed covert security with deterrence factor ϵ according to Definition 1.*

We formally prove Theorem 2 in the full version of this paper.

8 Evaluation

In order to evaluate the practicability of our protocols, i.e., to show that the judging party can be realized efficiently via a smart contract, we implemented

the judge of our third transformation (cf. Section 7) for the Ethereum blockchain and measured the associated execution costs. We focus on the third setting, the verification of protocols with a private transcript, since we expect this scenario to be the most expensive one due to the interactive punishment procedure. Further, we have extended the transformation such that the protocol does not require a public transcript of state hashes.

Our implementation includes the efficiency features described in the full version of this paper. In particular, we model the calculation of each round’s and party’s `computeRound` function as an arithmetic circuit and compress disputed calculations and messages using Merkle trees. The latter are divided into 32-byte chunks which constitute the leaf of the Merkle tree. The judge only needs to validate either the computation of a single arithmetic gate or the correctness of a single message chunk of a sent or received message together with the corresponding Merkle tree proofs. The proofs are logarithmic in the size of the computation resp. the size of a message. Messages are validated by defining a mapping from each chunk to a gate in the corresponding `computeRound` function.

In order to avoid redundant deployment costs, we apply a pattern that allows us to deploy the contract code just once and for all and create new independent instances of our FBC protocol without deploying further code. When starting a new protocol instance, parties register the instance at the existing contract which occupies the storage for the variables required by the new instance, e.g., the set of involved parties. Further, we implement the judge to be agnostic to the particular semi-honest protocol executed by the parties – recall that our FBC protocol wraps around a semi-honest protocol that is subject to the cut-and-choose technique. Every instance registered at the judge can involve a different number of parties and define its own semi-honest protocol. This means that the same judge contract can be used for whatever semi-honest protocol our FBC protocol instance is based on, e.g., for both the generation of Beaver triples and garbled circuits. Parties simply define for each involved party and each round the `computeRound` function as a set of gates, aggregate all gates into a Merkle tree and submit the tree’s root upon instance registration.

We perform all measurements on a local test environment. We setup the local Ethereum blockchain with *Ganache* (core version 2.13.2) on the latest supported hard fork, Muir Glacier. The contract is compiled to EVM byte code with *solc* (version 0.8.1, optimized on 20 runs). As common, we measure the efficiency of the smart contracts via its gas consumption – this metric directly translates to execution costs. Further, we estimate USD costs based on the prices (gas to ETH and ETH to USD) on Aug. 20, 2021 [Eth21, Coi21]. For comparison, a simple Ether transfer costs 21,000 gas resp. 2,81 USD.

In Table 1, we display the costs of the deployment, the registration of a new instance and the optimistic execution without any disputes. The costs of these steps only depend on the number of parties. In Table 2, we display the worst-case costs of a protocol execution for different protocol parameters, i.e., complexity of the `computeRound` functions, message size, communication rounds and number of parties. In order to determine the worst-case costs, we measured

Table 1: Costs for deployment, instance registration and optimistic execution.

Protocol steps	n	Cost	
		Gas	USD
Deployment		4 775 k	639.91
New instance	2	287 k	38.41
New instance	3	308 k	41.30
New instance	5	351 k	47.05
New instance	10	458 k	61.43
Honest execution	2	178 k	23.92
Honest execution	3	224 k	30.07
Honest execution	5	316 k	42.38
Honest execution	10	546 k	73.14

Gates: Number of gates in the circuit of each `computeRound` function.

Chunks: Number of chunks in each message.

R: Number of communication rounds.

n: Number of parties.

Table 2: Worst-case execution costs.

Gates	Chunks	R	n	Cost	
				Gas	USD
10	10	10	3	1 780 k	238.58
1 000	10	10	3	2 412 k	323.25
1 M	10	10	3	3 512 k	470.55
1 B	10	10	3	4 782 k	640.75
1 T	10	10	3	6 182 k	828.35
10	10	10	3	1 785 k	239.14
100	100	10	3	2 086 k	279.61
1 000	1 000	10	3	2 422 k	324.55
100	10	10	3	2 081 k	278.91
100	10	10	4	2 223 k	297.86
100	10	10	7	2 442 k	327.29
100	10	10	10	2 659 k	356.34
100	10	10	50	4 764 k	638.35
100	10	3	3	1 878 k	251.65
100	10	10	3	2 074 k	277.88
100	10	100	3	2 403 k	322.04
100	10	1 000	3	2 834 k	379.79

different dispute patterns, e.g., disputing sent messages or disputing gates of the `computeRound` functions, and picked the pattern with the highest costs. The execution costs, both optimistic and worst case, incorporate all protocol steps, incl. the secure funding of the instance. We exclude the derivation of the initial seeds as this step strongly depends on the underlying PVC protocol.

In the optimistic case, the costs of executing our protocol are similar to the ones of [ZDH19]. The authors report a gas consumption of 482 k gas while our protocol consumes between 465 k and 1 M gas, depending on the number of parties – recall that the protocol of [ZDH19] is restricted to the two-party setting. This overhead in our protocol when considering more than two parties is mainly introduced by the fact that [ZDH19] does assume a single deposit while our implementation requires each party to perform a deposit.

Unfortunately, we cannot compare worst-case costs directly, as the protocol of [ZDH19] validates the consistency of a fixed data structure, i.e., a garbled circuit, while our implementation validates the correctness of the whole protocol execution. In particular, [ZDH19] performs a bisection over the garbled circuit while we perform two bisections, first over the message history and then over the computation generating the outgoing messages; such a message might for example be a garbled circuit. Further, [ZDH19] focuses on a boolean circuit, while we model the `computeRound` function as an arithmetic circuit – as the EVM always stores data in 32-byte words, it does not make sense to model the function as a boolean circuit. Although not directly comparable, we believe the protocol of [ZDH19] to be more efficient for the special case of a two-party garbling protocol, as the protocol can exploit the fact that a dispute is restricted to a single message, i.e., the garbled circuit, and the data structure of this message is fixed such that the dispute resolution can be optimized to said data structure.

Our measurements indicate that the worst-case costs of each scenario are always defined by a dispute pattern that does not dispute a message chunk but a gate of the `computeRound` functions. This is why the message chunks have no

influence on the worst-case execution costs. Of course, this observation might be violated if we set the number of chunks much higher than the number of gates. However, it does not make sense to have more message chunks than gates because each message chunk needs to be mapped to a gate of the `computeRound` function defining the value of said chunk.

Both, the number of rounds and the number of parties increase the maximal size of the disputed message history and, hence, the depth of the bisected history tree. As the depth of the bisected tree grows logarithmic in the tree size, our protocol is highly scalable in the number of parties and rounds.

Finally, we note that we understand our implementation as a research prototype showing the practicability of our protocol. We are confident that additional engineering effort can further reduce the gas consumption of our contract.

Acknowledgments

The first, third, and fourth authors were supported by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 – 236615297 (CROSSING Project S7)*, by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*, and by Robert Bosch GmbH, by the Economy of Things Project. The second author was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister’s Office, and by ISF grant No. 1316/18.

References

- [ADMM14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *IEEE SP*, 2014.
- [AL07] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, 2007.
- [AO12] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In *ASIACRYPT*, 2012.
- [BK14] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO*, 2014.
- [BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, 1990.
- [Coi21] CoinMarketCap. Ethereum (ETH) price. <https://coinmarketcap.com/currencies/ethereum/>, 2021.
- [CRR11] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In *CCS*, 2011.
- [DKL⁺13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, 2013.





- [DOS20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. In *CRYPTO*, 2020.
- [DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multi-party computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.
- [EFS20] Lisa Eockey, Sebastian Faust, and Benjamin Schlosser. Optiswap: Fast optimistic fair exchange. In *ASIA CCS*, 2020.
- [Eth21] Etherscan. Ethereum Average Gas Price Chart. <https://etherscan.io/chart/gasprice>, 2021.
- [FHKS21] Sebastian Faust, Carmit Hazay, David Kretzler, and Benjamin Schlosser. Generic compiler for publicly verifiable covert multi-party computation. In *EUROCRYPT*, 2021.
- [GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, 1987.
- [HKK⁺19] Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wen-jie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. In *EUROCRYPT*, 2019.
- [KB14] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *CCS*, 2014.
- [KGC⁺18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *USENIX Security*, 2018.
- [KM15] Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. In *ASIACRYPT*, 2015.
- [SSS21] Peter Scholl, Mark Simkin, and Luisa Siniscalchi. Multiparty computation with covert security and public verifiability. *IACR Cryptol. ePrint Arch.*, 2021.
- [TR19] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *CoRR*, abs/1908.04756, 2019.
- [W⁺14] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.
- [WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, 2017.
- [WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.
- [YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In *CCS*, 2020.
- [ZDH19] Ruiyu Zhu, Changchang Ding, and Yan Huang. Efficient publicly verifiable 2pc over a blockchain with applications to financially-secure computations. In *CCS*, 2019.

C. Putting the Online Phase on a Diet: Covert Security from Short MACs

This chapter corresponds to the following publication. The full version is available at [89].

- [90] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Putting the Online Phase on a Diet: Covert Security from Short MACs”. In: *Topics in Cryptology - CT-RSA 2023 - Cryptographers’ Track at the RSA Conference 2023, San Francisco, CA, USA, April 24-27, 2023, Proceedings*. 2023, pp. 360–386. **Part of this thesis.**

Putting the Online Phase on a Diet: Covert Security from Short MACs

Sebastian Faust¹ , Carmit Hazay² , David Kretzler¹ , and Benjamin Schlosser¹ 

¹ Technical University of Darmstadt, Germany

`{first.last}@tu-darmstadt.de`

² Bar-Ilan University, Israel

`carmit.hazay@biu.ac.il`

Abstract. An important research direction in secure multi-party computation (MPC) is to improve the efficiency of the protocol. One idea that has recently received attention is to consider a slightly weaker security model than full malicious security – the so-called setting of *covert security*. In covert security, the adversary may cheat but only is detected with certain probability. Several works in covert security consider the offline/online approach, where during a costly offline phase correlated randomness is computed, which is consumed in a fast online phase. State-of-the-art protocols focus on improving the efficiency by using a covert offline phase, but ignore the online phase. In particular, the online phase is usually assumed to guarantee security against malicious adversaries. In this work, we take a fresh look at the offline/online paradigm in the covert security setting. Our main insight is that by weakening the security of the online phase from malicious to covert, we can gain significant efficiency improvements during the offline phase. Concretely, we demonstrate our technique by applying it to the online phase of the well-known TinyOT protocol (Nielsen et al., CRYPTO '12). The main observation is that by reducing the MAC length in the online phase of TinyOT to t bits, we can guarantee covert security with a detection probability of $1 - \frac{1}{2^t}$. Since the computation carried out by the offline phase depends on the MAC length, shorter MACs result in a more efficient offline phase and thus speed up the overall computation. Our evaluation shows that our approach reduces the communication complexity of the offline protocol by at least 35% for a detection rate up to $\frac{7}{8}$. In addition, we present a new generic composition result for analyzing the security of online/offline protocols in terms of concrete security.

Keywords: Multi-Party Computation (MPC) · Covert Security · Offline/Online · Deterrence Composition

1 Introduction

Secure multi-party computation (MPC) allows a set of distrusting parties to securely compute an arbitrary function on private inputs. While originally MPC was mainly studied by the cryptographic theory community, in recent years many industry applications have been envisioned in areas such as machine learning [KVH⁺21], databases [VSG⁺19], blockchains [Zen] and more [ABL⁺18, MPC]. One of the main challenges for using MPC protocols in practice is their huge overhead in terms of efficiency. Over the last decade, tremendous progress has been made both on the protocol side as well as the engineering level to move MPC protocols closer to practice [DPSZ12, DKL⁺13, KOS16, KPR18, BCS19, CKR⁺20, Ors20].

Most efficient MPC protocols work in the honest-but-curious setting. In this setting, the adversary must follow the protocol specification but tries to learn additional information from the interaction with the honest parties. A much stronger security notion is to consider malicious security, where the corrupted parties may arbitrarily deviate from the specification in order to attack the protocol. Unfortunately, however, achieving malicious security is much more challenging and typically results into significant efficiency penalties [KOS16, DILO22].

An attractive middle ground between the efficient honest-but-curious model and the costly malicious setting is *covert security* originally introduced by Aumann and Lindell [AL07]. As in malicious security, the adversary may attack the honest parties by deviating arbitrarily from the protocol specification but may get detected in this process. Hence, in contrast to malicious security such protocols do not prevent cheating, but instead de-incentivize malicious behavior as an adversary may fear getting caught. The latter may lead to reputational damage or financial punishment, which for many real-world settings is a sufficiently strong countermeasure against attacks. Moreover, since covert security does not need to prevent cheating at the protocol level, it can lead to significantly improved efficiency. Let us provide a bit more detail on how to construct covert secure protocols.

The cut-and-choose technique. In a nutshell, all known protocols with covert security amplify the security of a semi-honest protocol by applying the cut-and-choose technique. In this technique, the semi-honest protocol is executed t times where $t - 1$ of the executions are checked for correctness via revealing their entire private values. The remaining unchecked instance stays hidden and thus can be used for computing the output. Since in the protocol the $t - 1$ checked instances are chosen uniformly at random, any cheating attempt is detected with probability at least $\frac{t-1}{t}$, which is called the *deterrence factor* of the protocol and denoted by ϵ . The overhead of the cut-and-choose approach is roughly a factor t compared to semi-honest protocols due to the execution of t semi-honest instances.

The offline/online paradigm. An important technique to construct efficient MPC protocols is to split the computation in an input independent offline phase and an input dependent online phase. The goal of this approach is to shift

the bulk of the computational effort to the offline phase such that once the private inputs become available the evaluation of the function can be done efficiently. To this end, parties pre-compute correlated randomness during the offline phase, which is consumed during the online phase to speed up the computation. Examples for offline/online protocols are SPDZ [DPSZ12], authenticated garbling [WRK17a, WRK17b] and the TinyOT approach [NNOB12, LOS14, BLN⁺21, FKOS15].

While traditionally the offline/online paradigm has been instantiated either in the honest-but-curious or malicious setting, several recent works have considered how to leverage the offline/online approach to speed-up covert secure protocols [DKL⁺13, DOS20, FHKS21]. The standard approach is to take a covertly secure offline phase and combine it with a maliciously secure online phase. Since the offline phase is most expensive, this results into a significant efficiency improvement. Moreover, since the offline phase is input independent, it is particularly well suited for the cut-and-choose approach used for constructing covert secure protocols. In contrast to the offline phase, for the online phase we typically rely on a maliciously secure protocol. The common belief is that the main efficiency bottleneck is the offline phase, and hence optimizing the online phase to achieve covert security (which is also more challenging since we need to deal with the private inputs) is of little value. In our work, we challenge this belief and study the following question:

Can we improve the overall efficiency of a covertly secure offline/online protocol by relaxing the security of the online phase to covert security?

1.1 Contribution

Our main contribution is to answer the above question in the affirmative. Concretely, we show that significant efficiency improvements are possible by switching from a maliciously secure online phase to covert security.

To this end, we introduce a new paradigm to achieve covert security. Instead of amplifying semi-honest security using cut-and-choose, we start with a maliciously secure protocol and weaken its security. In malicious security, successful cheating of the adversary is only possible with negligible probability in the statistical security parameter. For protocol instantiations, this parameter is typically set to 40. The core idea is to show that in the setting of covert security, we can significantly reduce the value of the statistical security parameter *without* losing in security. We are the first to describe this new method of achieving covert security by weakening malicious security.

For achieving covert security of already efficient online protocols, the naive cut-and-choose approach is not a viable option due to its inherent overhead. In contrast, our approach is particularly interesting for these protocols. In addition, we observe that for several offline/online protocols, a reduction to covert security in the online phase reduces the amount of precomputation required. This results in an overall improved efficiency.

To illustrate the benefits of our paradigm, we apply it to the well-known TinyOT [NNOB12] protocol for two-party computation for boolean circuits based on the secret-sharing approach. This protocol is a good benchmark for oblivious transfer (OT)-based protocols and hasn't been considered before for the covert setting. The original TinyOT protocol consists of a maliciously secure offline and online phase where MACs ensure the correctness of the computation performed during the online phase. While the efficiency of the offline phase can be improved by making this phase covertly secure using the cut-and-choose approach, we apply our paradigm to the online phase to gain additional efficiency improvements. Our insight is that instead of using 40-bit MACs, which is typically done for an actively secure online phase, using t -bits MACs results in a covertly secure online phase with deterrence factor $1 - \frac{1}{2^t}$. We formally prove the covert security of this online protocol.

As touched on earlier, shortening the MAC length of the TinyOT online phase has a direct impact on the computation overhead carried out in the offline phase. In particular, the size of the oblivious transfers that need to be performed depend on the MAC length and thus this number can be reduced. Concretely, we compare the communication complexity of a cut-and-choose-based offline phase for different choices of MAC lengths. We can show that the communication complexity of the offline protocol reduces by at least 35% for a deterrence factor up to $\frac{7}{8}$.

While we chose the TinyOT protocol for demonstrating our new paradigm, we can apply our techniques also for other offline/online protocols in the two- and multi-party case, e.g., [LOS14, BLN⁺21, FKOS15, WRK17a, WRK17b].

As a second major technical contribution, we show that the combination of a covert offline and covert online phase achieves the same deterrence factor as a covert offline phase combined with an active online phase. We show this result in a generic way by presenting a *deterrence replacement theorem*. Intuitively, when composing a covertly secure offline phase with a covertly secure online phase, the deterrence factor of the composed protocol needs to consider the worst deterrence of both phases. This is easy to see, since the adversary can always try to cheat in that phase where the detection probability is smaller. While easy at first sight, the formalization requires a careful analysis and adds restrictions on the class of protocols for which such composition can be shown. By applying our deterrence replacement theorem, we show for offline/online protocols that the overall detection probability is computed as the minimum of the detection probability of the offline phase and the detection probability of the online phase.

While this result was proven by Aumann and Lindell [AL07] for a weak notion of covert security, the *failed-simulation formulation*, we are the first to formally present a proof in the strongest setting of covert security which is also mostly used in the literature. The definitional framework of the failed-simulation formulation and the one of all of the stronger notions are fundamentally different. In particular, the failed-simulation formulation relies on the ideal functionality defined for the malicious setting but allows for failed simulations. The stronger notions define a covert ideal functionality explicitly capturing the properties

of the covert setting, i.e., the possible cheating attempts of the adversary. For this reason, it is not straightforward to translate the proof techniques from the failed-simulation formulation to the stronger notions.

1.2 Related Work

Short MACs. Hazay et al. [HOSS18] also considered short MAC keys for TinyOT, but in the context of concretely efficient large-scale MPC in the active security setting with a minority of honest parties. The main idea of their work is to distribute secret key material between all parties such that the security is based on the concatenation of all honest parties' keys. In contrast, we achieve more efficient covert security and the security is based on each party's individual key.

TinyOT extensions. In the two-party setting, the TinyOT protocol is extended by the TinyTables [DNNR17] and the MiniMac [DZ13] protocols. The former improves the online communication complexity by relying on precomputed scrambled truth tables. The precomputation of these works is based on the offline phase of TinyOT. Therefore, we believe that our techniques can be applied to the TinyTables protocol as well. We focus in our description on the original TinyOT protocol to simplify presentation.

The MiniMac protocol uses error correcting codes for authentication of bit vectors and is in particular interesting for “well-formed” circuits that allow for parallelization of computation. The sketched precomputation of MiniMac is based on the SPDZ-precomputation [DPSZ12]. In the SPDZ protocol, MACs represent field elements instead of binary strings as in TinyOT. Therefore, it is not straight-forward to apply our techniques to the MiniMac protocol. We leave it as an open question if our techniques can be adapted to this setting.

Larraia et al. and Burra et al. [LOS14, BLN⁺21] show how to extend TinyOT to the multi-party setting. Our paradigm can be applied to these protocols as well as to the precomputation of [FKOS15].

Authenticated garbling. The authenticated garbling protocols [WRK17a, WRK17b, KRRW18, YWZ20] achieve constant round complexity and active security by utilizing an authenticated garbled circuit. For authentication, the protocols rely on a TinyOT-style offline phase. Hence, we believe that our approach can improve the efficiency of the authenticated garbling protocols as well (when moving to the setting of covert security).

Arithmetic computation. The family of SPDZ protocols [DPSZ12, DKL⁺13, KOS16, KPR18, CDE⁺18] provide means to perform multi-party computation with active security on arithmetic circuits. Damgård et al. [DKL⁺13] have already considered the covert setting but only reduced the security of the offline phase to covert security. As already mentioned above in the context of MiniMac, we leave it as an interesting open question to investigate if our approach can be translated to the arithmetic setting of the SPDZ family in which MACs are represented as field elements.

Pseudorandom Correlation Generators. Recently, pseudorandom correlation generators (PCGs) were presented to compute correlated randomness with sublinear communication [BCG⁺19, BCG⁺20a, BCG⁺20b]. While this is a promising approach, efficient constructions are based on variants of the learning parity with noise (LPN) assumption. These assumptions are still not fully understood, especially compared to oblivious transfer which is the base of TinyOT.

1.3 Technical Overview

Notions of covert security. The notion of *covert security with ϵ -deterrence factor* was proposed by Aumann and Lindell in 2007 [AL07], who introduced a hierarchy of three different variants. The weakest variant is called the *failed-simulation formulation*, the next stronger is the *explicit cheat formulation (ECF)* and the strongest variant is the *strong explicit cheat formulation (SECF)*. The last is also the most widely used variant of covert security. In the failed-simulation formulation, the adversary is able to cheat depending on the honest parties' inputs. This undesirable behavior is prevented in the stronger variants. In the ECF notion, the adversary learns the inputs of the honest parties even if cheating is detected. Finally, SECF prevents the adversary from learning anything in case cheating is detected.

In this work, we introduce on a new notion that lies between ECF and SECF. We call it *intermediate explicit cheat formulation (IECF)* (cf. Section 2), where we let the adversary learn the outputs of the corrupted parties even if cheating is detected. This is a strictly stronger security guarantee than ECF, where the adversary also learns the inputs of the honest parties. Our new notion captures protocols where an adversary learns its own outputs (which may depend on honest parties inputs) before the honest parties detect cheating. However, we emphasize that the adversary cannot prevent detection by the honest parties. In particular, it must make its decision on whether to cheat or not before learning its outputs. Moreover, notice that in case when the adversary does not cheat, it would anyway learn these outputs, and hence IECF is only a very mild relaxation of the SECF notion.

Composition of covert protocol. Composition theorems allow to modularize security proofs of protocols and thus are tremendously useful for protocol design. Aumann and Lindell presented two sequential composition theorems for protocols in the covert security model [AL07]. One for the failed-simulation formulation and one for the (S)ECF. In the following, we focus on the later theorem since these notions are closer to the IECF notion. The composition theorem presented in [AL07] allows to analyze the security of a protocol in a hybrid model where the parties have access to hybrid functionalities. In more detail, the theorem states that a protocol that is covertly secure with deterrence factor ϵ in a hybrid model where parties have access to a polynomial number of functionalities, which themselves have deterrence factors, then the protocol is also secure if the hybrid functionalities are replaced with protocols realizing the functionalities with the corresponding deterrence values. Note that the theorem states that a

composed protocol using subprotocols instead of hybrid functionalities has the same deterrence factor as when analyzed with (idealized) hybrid functionalities.

Aumann and Lindell’s theorem is very useful to show security of a complex protocol. Unfortunately, however, the theorem of Aumann and Lindell does not make any statement how the deterrence factor of hybrid functionalities influences the deterrence factor of the overall protocol. Instead, the deterrence factor of the overall protocol has to be determined depending on the concrete deterrence factors of the hybrid functionalities. We are looking for a composition theorem that goes one step further. In particular, we develop a theorem that allows to analyze a protocol’s security and its deterrence factor in a simple model where no successful cheating in hybrid functionalities is possible, i.e., a deterrence factor of $\epsilon = 1$. Then, the theorem should help deriving the deterrence factor of the composed protocol when cheating in hybrid functionalities is possible with a fixed probability, i.e., $\epsilon < 1$.

Deterrence replacement theorem. Our deterrence replacement theorem fills the aforementioned gap (cf. Section 3). Let Hy_1 and Hy_2 be two hybrid worlds. In Hy_1 an offline functionality exists with deterrence factor 1. In Hy_2 the same offline functionality has deterrence factor ϵ_{off}^* . Our theorem states that a protocol, which is covertly secure with deterrence factor ϵ_{on} in Hy_1 , is covertly secure with deterrence factor $\epsilon_{\text{on}}^* := \text{Min}(\epsilon_{\text{on}}, \epsilon_{\text{off}}^*)$ in Hy_2 . While we have to impose some restrictions on the protocols that our theorem can be applied on, practical offline/online protocols [DPSZ12, NNOB12, WRK17a, WRK17b] fulfill these restrictions or can easily be adapted to do so. The main benefit of our theorem is to simplify the analysis of a protocol’s security by enabling the analysis in a model where successful cheating in the offline functionality does not occur. In addition, our theorem implies that the deterrence factor of the online phase can be as low as the deterrence factor of the offline phase without any security loss.

Achieving covert security. Most covertly secure protocols work by taking a semi-honest secure protocol and applying the cut-and-choose technique. In contrast, we present a new approach to achieve covert security where instead of amplifying semi-honest security, we downgrade malicious security. Our core idea is to obtain covert security by reducing the statistical security parameter of a malicious protocol.

As highlighted in the contribution, reducing the security of the online phase to covert has the potential to improve the efficiency of the overall protocol execution. This improvement does not come from a speed-up in the online phase, in fact the online phase can become slightly less efficient, but from lower requirements on the offline phase. Using the cut-and-choose approach to get a covertly secure online phase incurs an overhead to the semi-honest protocol that is linear in the number of executed instances. This overhead might exceed the efficiency gap between the semi-honest and the malicious protocol rendering the cut-and-choose-based covert offline phase significantly less efficient than the malicious online phase. In this case, the overhead of the online phase can vanish the gains of the faster offline phase. In contrast, our approach comes with a small constant overhead to the malicious protocol such that the overall efficiency gain is

preserved. This makes our approach particularly interesting for actively secure protocols that are already very efficient such as information-theoretic online protocols, e.g., the online phase of TinyOT [NNOB12].

The TinyOT protocol. We illustrate the benefit of our new paradigm for achieving covert security by applying it to the maliciously secure online phase of TinyOT [NNOB12]. We start with a high-level overview of TinyOT.

The TinyOT protocol is a generic framework for computing Boolean circuits based on the secret sharing paradigm for two-party computation. The protocol splits the computation into an offline and an online phase. In the offline phase, the parties compute authenticated bits and authenticated triples. For instance, the authentication of a bit x known to a party \mathcal{A} is achieved by having the other party \mathcal{B} hold a global key $\Delta_{\mathcal{B}}$, a random t -bit key $K[x]$, and having \mathcal{A} hold the bit x and a t -bit MAC $M[x] = K[x] \oplus x \cdot \Delta_{\mathcal{B}}$. In the online phase, parties evaluate the circuit with secret-shared wire values where each share is authenticated given the precomputed data. Due to the additive homomorphism of the MACs, addition gates can be computed non-interactively. For each multiplication gate, the parties interactively compute the results by consuming a precomputed multiplication triple. At the end of the circuit evaluation, a party learns its output, i.e., the value of an output wire, by receiving the other party’s share on that wire. The correct behavior of all parties is verified by checking the MACs on the output wire shares.

Covert online protocol. The authors of TinyOT showed that successfully breaking security of the online phase is equivalent to guessing the global MAC key of the other party. In this work, we translate this insight to the covert setting. In particular, we show that the online phase of a TinyOT-like protocol with a reduced MAC length of t -bits implements covert security with a deterrence factor of $1 - (\frac{1}{2})^t$ (cf. Section 4).

The resulting protocol can be modified with small adjustments to achieve all known notions of covert security. In particular, the unmodified version of TinyOT implements a variant of covert security in which the adversary learns the output of the protocol, and, only then, decides on its cheating attempt. We achieve the IECF, i.e., the notion in which the adversary always learns the output of the corrupted parties, even in case of detected cheating, by committing to the outputs bits and MACs before opening them. Due to the commitments, the adversary needs to decide first if it wants to cheat and only afterwards it learns the output. However, since the adversary receives the opening on the commitment of the honest party first, it learns the output even if it committed to incorrect values or refuses to open its commitment, both of which are considered cheating. Finally, in order to achieve the SECF, we have to prevent the adversary from inserting incorrect values into the commitment. We can do so by generating the commitments as part of the function whose circuit is evaluated. Only after the parties checked both, correct behavior throughout the evaluation and correctness of the received outputs, i.e., the commitments, the parties exchange the openings of the commitments. This way, we ensure that the adversary only receives its output if it behaved honestly or cheated successfully which fulfills the SECF.

In this work, we focus on the IECF. On one hand, we assess the IECF to constitute a minor loss of security compared to the SECF. This is due to the fact that we are in the security-with-abort setting, implying that the honest parties already approve the risk of giving the adversary its output while not getting an output themselves. On the other hand, the efficiency overhead of the IECF compared to the weaker variant of covert achieved by the unmodified protocol just consists out of a single commit-and-opening step accounting for 48 bytes per party (if instantiated via a hash function and with 128 bit security). In contrast, the SECF requires generating the commitments as part of the circuit which incurs a much higher efficiency overhead. Therefore, we assess the protocol achieving the IECF notion to depict a much better trade-off between efficiency overhead and security loss than the other notions.

Evaluation. Our result shows that we can safely reduce the security level of the online phase without compromising on the security of the overall protocol. As we show in the evaluation section (cf. Section 5), this improves the efficiency of the overall protocol. Concretely, the main improvements come from savings during the offline phase since using our techniques the online phase gets less demanding by relying on shorter MACs. We quantify these improvements by evaluating the communication complexity of the offline phase depending on the length of the generated MACs. More precisely, when using an actively secure online phase, the MAC length needs to be 40 Bits, while for achieving covert security, we can set the length of the MACs to a significantly lower value t . This results into a deterrence factor of $1 - \frac{1}{2^t}$. Our evaluation shows that we can reduce the communication complexity of the offline protocol by at least 35% for a deterrence factor of up to $\frac{7}{8}$.

2 Covert Security

A high-level comparison between the notions of covert security presented by Aumann and Lindell [AL07] is stated in Section 1.3. Next, we present details about the *explicit cheat formulation (ECF)* and the *strong explicit cheat formulation (SECF)*. Afterwards, we present our new notion which lies strictly between the ECF and the SECF.

The ECF and the SECF consider an ideal functionality where the adversary explicitly sends a `cheati` command for the index i of a corrupted party to the functionality which then decides if cheating is detected with probability ϵ . In the ECF, the adversary learns the honest parties' inputs even if cheating is detected, which is prevented by the SECF. In addition, the adversary can also send a `corruptedi` or `aborti` command, which is forwarded to the honest parties. The `corruptedi` command models a blatant cheat option, where the adversary cheats in a way that will always be detected, and the `aborti` command models an abort of a corrupted party. Later, Faust et al. [FHKS21] proposed to extract the *identifiable abort* property as it can be considered orthogonal and of independent interest (cf. [IOZ14]). For the covert notion, this means that if a corrupted party

aborts, the ideal functionality only sends `abort` to the honest parties instead of `aborti` for i being the index of the aborting party.

In the following, we present a new notion for covert security called the *intermediate explicit cheat formulation (IECF)*. We follow the approach of [FHKS21] and present our notion without the identifiable abort property. In addition, we clean up the definition by merging the blatant cheat option, where cheating is always detected, with the cheat attempt that is only detected with a fixed probability. To this end, if the adversary sends the `cheat`-command, we allow the adversary to specify any detection probability between the deterrence factor and 1. Furthermore, we enable the adversary to force a cheating detection or abort even if the ideal functionality signals undetected cheating. This additional action does not provide further benefit to the adversary and thus does not harm the security provided by our notion. Since the decision solely depends on the adversary, the change also does not restrict the adversary.

Finally, and most important, our notion allows the adversary to learn the outputs of the corrupted parties but nothing else if cheating is detected. Therefore, it lies between the ECF, where the adversary learns the inputs of all parties even if cheating is detected, and the SECF, where the adversary learns nothing if cheating is detected. Since our notion is strictly between the ECF and the SECF, we call it the IECF.

Next, we present the IECF in full details in the following and state the difference to the SECF afterwards.

Intermediate explicit cheat formulation. As in the standalone model, the notions are defined in the real world/ideal world paradigm. This means, the security of a protocol is shown by comparing the real-world execution with an ideal-world execution. In the *real world*, the parties jointly compute the desired function f using a protocol π . Let n be the number of parties and let $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$, where $f = (f_1, \dots, f_n)$ is the function computed by π . We define for every vector of inputs $\bar{x} = (x_1, \dots, x_n)$ the vector of outputs $\bar{y} = (f_1(\bar{x}), \dots, f_n(\bar{x}))$ where party P_i with input x_i obtains the output $f_i(\bar{x})$. During the execution of π , the adversary Adv can corrupt a subset $\mathcal{I} \subset [n]$ of all parties. We define $\text{REAL}_{\pi, \text{Adv}(z), \mathcal{I}}(\bar{x}, 1^\kappa)$ as the output of the protocol execution π on input $\bar{x} = (x_1, \dots, x_n)$ and security parameter κ , where Adv on auxiliary input z corrupts parties \mathcal{I} . We further specify $\text{OUTPUT}_i(\text{REAL}_{\pi, \text{Adv}(z), \mathcal{I}}(\bar{x}, 1^\kappa))$ to be the output of party P_i for $i \in [n]$.

In contrast, in the *ideal world*, the parties send their inputs to the uncorruptible ideal functionality \mathcal{F} which computes function f and returns the result. Hence, the computation in the ideal world is correct by definition. The security of π is analyzed by comparing the ideal-world execution with the real-world execution. The ideal world in covert security is slightly changed compared to the standard model of secure computation. In particular, in covert security, the ideal world allows the adversary to cheat, and cheating is detected at least with some fixed probability ϵ which is called the *deterrence factor*. Let $\epsilon : \mathbb{N} \rightarrow [0, 1]$ be a function. The execution in the ideal world in our *IECF* notion is defined as follows:

Inputs: Each party obtains an input, where the i^{th} party's input is denoted by x_i . We assume that all inputs are of the same length and call the vector $\bar{x} = (x_1, \dots, x_n)$ balanced in this case. The adversary receives an auxiliary input z . In case there is no input, the parties will receive $x_i = \text{ok}$.

Send inputs to ideal functionality: Any honest party P_j sends its received input x_j to the ideal functionality. The corrupted parties, controlled by ideal world adversary \mathcal{S} , may either send their received input, or send some other input of the same length to the ideal functionality. This decision is made by \mathcal{S} and may depend on the values x_i for $i \in \mathcal{I}$ and the auxiliary input z . Denote the vector of inputs sent to the ideal functionality by \bar{x} . In addition, \mathcal{S} can send a special cheat or abort message w .

Abort options: If \mathcal{S} sends $w = \text{abort}$ to the ideal functionality as its input, then the ideal functionality sends **abort** to all honest parties and halts.

Attempted cheat option: If \mathcal{S} sends $w = (\text{cheat}_i, \epsilon_i)$ for $i \in \mathcal{I}$ and $\epsilon_i \geq \epsilon$, the ideal functionality proceeds as follows:

1. With probability ϵ_i , the ideal functionality sends **corrupted_i** to all honest parties. In addition, the ideal functionality computes $(y_1, \dots, y_n) = f(\bar{x})$ and sends **(corrupted_i, {y_j}_{j ∈ I})** to \mathcal{S} .
2. With probability $1 - \epsilon_i$, the ideal functionality sends **undetected** to \mathcal{S} along with the honest parties' inputs $\{x_j\}_{j \notin \mathcal{I}}$. Then, \mathcal{S} sends output values $\{y_j\}_{j \notin \mathcal{I}}$ of its choice for the honest parties to the ideal functionality. Then, for every $j \notin \mathcal{I}$, the ideal functionality sends y_j to P_j . The adversary may also send **abort** or **corrupted_i** for $i \in \mathcal{I}$, in which case the ideal functionality sends **abort** or **corrupted_i** to every P_j for $j \notin \mathcal{I}$.

The ideal execution ends at this point. Otherwise, if no w equals **abort** or **(cheat_i, ·)** the ideal execution proceeds as follows.

Ideal functionality answers adversary: The ideal functionality computes $(y_1, \dots, y_n) = f(\bar{x})$ and sends y_i to \mathcal{S} for all $i \in \mathcal{I}$.

Ideal functionality answers honest parties: After receiving its outputs, the adversary sends **abort**, **corrupted_i** for some $i \in \mathcal{I}$, or **continue** to the ideal functionality. If the ideal functionality receives **continue** then it sends y_j to all honest parties P_j ($j \notin \mathcal{I}$). Otherwise, if it receives **abort** resp. **corrupted_i**, it sends **abort** resp. **corrupted_i** to all honest parties.

Outputs: An honest party always outputs the message it obtained from the ideal functionality. The corrupted parties output nothing. The adversary \mathcal{S} outputs any arbitrary (probabilistic polynomial-time computable) function of the initial inputs $\{x_i\}_{i \in \mathcal{I}}$, the auxiliary input z , and the messages obtained from the ideal functionality.

We denote by $\text{IDEALC}_{f, \mathcal{S}(z), \mathcal{I}}^{\epsilon}(\bar{x}, 1^{\kappa})$ the output of the honest parties and the adversary in the execution of the ideal model as defined above, where \bar{x} is the input vector and the adversary \mathcal{S} runs on auxiliary input z .

Definition 1 (Covert security - intermediate explicit cheat formulation). Let f, π , and ϵ be as above. A protocol π securely computes f in the

presence of covert adversaries with ϵ -deterrence *if for every non-uniform probabilistic polynomial-time adversary Adv in the real world, there exists a non-uniform probabilistic polynomial-time adversary S for the ideal model such that for every $\mathcal{I} \subseteq [n]$, every balanced vector $\bar{x} \in (\{0, 1\}^*)^n$, and every auxiliary input $z \in \{0, 1\}^*$:*

$$\{\text{IDEALC}_{f, \mathcal{S}(z), \mathcal{I}}^c(\bar{x}, 1^\kappa)\}_{\kappa \in \mathbb{N}} \stackrel{c}{\equiv} \{\text{REAL}_{\pi, \text{Adv}(z), \mathcal{I}}(\bar{x}, 1^\kappa)\}_{\kappa \in \mathbb{N}}$$

The *SECF* notions follows the *IECF* notion with one single change. Instead of sending $(\text{corrupted}_i, \{y_j\}_{j \in \mathcal{I}})$ to \mathcal{S} in case of detected cheating, the ideal functionality only sends (corrupted_i) . This means that in the *SECF* the ideal adversary does not learn the output of corrupted parties in case cheating is detected.

3 Offline/Online Deterrence Replacement

Offline/online protocols split the computation of a function f into two parts. In the offline phase, the parties compute correlated randomness independent of the actual inputs to f . In the online phase, the function f is computed on the private inputs of all parties while the correlated randomness from the offline phase is consumed to accelerate the execution. When considering covert security, the adversary may cheat in both the offline and the online phase. The cheating detection probability might differ in these two phases. Intuitively, the deterrence factor of the overall protocol needs to consider the worst-case detection probability. This is easy to see, since the adversary can always choose to cheat during that phase where the detection probability is smaller.

While the above is easy to see at a high level, the outlined intuition needs to be formally modeled and proven. We take the approach of describing offline/online protocols within a hybrid model. This means, the offline phase is formalized as a hybrid functionality to which the adversary can signal a cheat attempt. This hybrid functionality is utilized by the online protocol during which the adversary can cheat, too. We formally describe the hybrid model in Section 3.1.

Next, we present our offline/online deterrence replacement theorem in Section 3.2. Let π_{on} be an online protocol that is covertly secure with deterrence factor ϵ_{on} while any cheat attempt during the offline phase is detected with probability $\epsilon_{\text{off}} = 1^3$. Then, our theorem shows that if the detection probability during the offline phase is reduced to $\epsilon'_{\text{off}} < 1$, π_{on} is also covertly secure with a deterrence factor of $\epsilon'_{\text{on}} = \min(\epsilon_{\text{on}}, \epsilon'_{\text{off}})$. This means, the new deterrence factor is the minimum of the detection probability of the old online protocol, in which successful cheating during the offline phase is not possible, and the detection probability of the new offline phase. Intuitively, our theorem quantifies the effect on the deterrence factor of the online protocol when replacing the deterrence

³ Covert security with deterrence factor 1 can be realized by a maliciously secure protocol as shown by Asharov and Orlandi [AO12].

factor of the offline hybrid functionality with a different value. This is why we call Theorem 1 the deterrence replacement theorem.

The main purpose of our theorem is to allow the analysis of the security of an online protocol in a simple setting where $\epsilon_{\text{off}} = 1$. Since in this setting cheating during the offline phase is always detected, the security analysis and the calculation of the online deterrence factor ϵ_{on} are much simpler. Once the security of π_{on} has been proven in the hybrid world, in which the offline phase is associated with deterrence factor 1, and ϵ_{on} has been determined, our theorem allows to derive security of π_{on} in the hybrid world, in which the offline phase is associated with deterrence factor ϵ'_{off} , and determines the deterrence factor to be $\epsilon'_{\text{on}} = \min(\epsilon'_{\text{off}}, \epsilon_{\text{on}})$.

While the effect of deterrence replacement was already analyzed by Aumann and Lindell [AL07] for a weak variant of covert security, we are the first to consider deterrence replacement in a widely adopted and strong variant of covert security. We discuss the relation to [AL07] in Appendix B.

3.1 The Hybrid Model

We consider a hybrid model to formalize the execution of offline/online protocols. Within such a model, parties exchange messages between each other but also have access to hybrid functionalities $\mathcal{F}_1, \dots, \mathcal{F}_\ell$. These hybrid functionalities work like trusted parties to compute specified functions. The hybrid model is thus a combination of the real model, in which parties exchange messages according to the protocol description, and the ideal model, in which parties have access to an idealized functionality.

A protocol in a hybrid model consists of standard messages sent between the parties and calls to the hybrid functionalities. These calls instruct the parties to send inputs to the hybrid functionality, which delivers back the output according to its specification. After receiving the outputs from the hybrid functionality, the parties continue the execution of the protocol. When instructed to send an input to the hybrid functionality, all honest parties follow this instruction and wait for the return value before continuing the protocol execution.

The interface provided by a hybrid functionality depends on the security model under consideration. Since we deal with covert security, the adversary is allowed to send special commands, e.g., `cheat`, to the hybrid functionality. In case the functionality receives `cheat` from the adversary, the functionality throws a coin to determine whether or not the cheat attempt will be detected by the honest parties. The detection probability is defined by the deterrence factor of this functionality. We use the notation \mathcal{F}_f^ϵ to denote a hybrid functionality computing function f with deterrence factor ϵ . The notation of a $(\mathcal{F}_{f_1}^{\epsilon_1}, \dots, \mathcal{F}_{f_\ell}^{\epsilon_\ell})$ -hybrid model specifies the hybrid functionalities accessible by the parties.

The hybrid model technique is useful to modularize security proofs. Classical composition theorems for passive and active security [Can00] as well as for covert security [AL07] build the foundation for this proof technique. Informally, these theorems state that if a protocol π is secure in the hybrid model where the

parties use a functionality \mathcal{F}_f and there exists a protocol ρ that securely realizes \mathcal{F}_f , then the protocol π is also secure in a model where \mathcal{F}_f is replaced with ρ .

3.2 Our Theorem

We start by assuming an online protocol π_{on} that realizes an online functionality $\mathcal{F}_{f_{\text{on}}}^{\epsilon_{\text{on}}}$ in the $\mathcal{F}_{f_{\text{off}}}^1$ -hybrid world. This means the deterrence factor of π_{on} is ϵ_{on} and the deterrence factor of the offline functionality is 1 which means that every cheating attempt in the offline phase will be detected. Next, our theorem states that replacing the deterrence factor 1 of the offline hybrid functionality with any $\epsilon'_{\text{off}} \in [0, 1]$ results in a deterrence factor of the online protocol of $\epsilon'_{\text{on}} = \min(\epsilon_{\text{on}}, \epsilon'_{\text{off}})$, i.e., the minimum of the previous deterrence factor of the online protocol and the new deterrence of the offline hybrid functionality.

Formally, we model the composition of an offline and an online phase via the hybrid model. Let $f_{\text{off}} : (\{\perp\}_{j \notin \mathcal{I}}, \{x_i^{\text{off}}\}_{i \in \mathcal{I}}) \rightarrow (y_1^{\text{off}}, \dots, y_n^{\text{off}})$ be an n -party probabilistic polynomial-time function representing the offline phase, where \mathcal{I} denotes the set of corrupted parties. We model the offline functionality in such a way that the honest parties provide no input, the adversary may choose the randomness used by the corrupted parties and the functionality produces outputs which depend on the randomness of the corrupted parties and further random choices. The n -party probabilistic polynomial-time online function is denoted by $f_{\text{on}} : (x_1, \dots, x_n) \rightarrow (y_1^{\text{on}}, \dots, y_n^{\text{on}})$. We use the abbreviation $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ and $\mathcal{F}_{\text{on}}^{\epsilon_{\text{on}}}$ for $\mathcal{F}_{f_{\text{off}}}^{\epsilon'_{\text{off}}}$ and $\mathcal{F}_{f_{\text{on}}}^{\epsilon_{\text{on}}}$.

Our composition theorem puts some restrictions on the online protocol π_{on} that we list below and discuss in more technical depth in Appendix A. First, we require that $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ is called only once during the execution of π_{on} and this call happens at the beginning of the protocol before any other messages are exchanged. Second, we require that if $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ returns `corruptedi` to the parties, then π_{on} instructs the parties to output `corruptedi`. Practical offline/online protocols [DPSZ12, NNOB12, WRK17a, WRK17b] either directly fulfill these requirements or can easily be adapted to do so. We are now ready to formally state our deterrence replacement theorem.

Theorem 1 (Deterrence replacement theorem). *Let f_{off} and f_{on} be n -party probabilistic polynomial-time functions and π_{on} be a protocol that securely realizes $\mathcal{F}_{\text{on}}^{\epsilon_{\text{on}}}$ in the $\mathcal{F}_{\text{off}}^1$ -hybrid model according to Definition 1, where f_{off} , f_{on} and π_{on} are defined as above. Then, π_{on} securely realizes $\mathcal{F}_{\text{on}}^{\epsilon'_{\text{on}}}$ in the $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ -hybrid model according to Definition 1, where $\epsilon'_{\text{on}} = \min(\epsilon_{\text{on}}, \epsilon'_{\text{off}})$.*

Remarks. Our theorem focuses on the offline/online setting where only a single hybrid functionality is present. Nevertheless, it can be extended to use additional hybrid functionalities with fixed deterrence factors. In addition, we present our theorem for the intermediate explicit cheat formulation to match the definition given in Section 2. We emphasize that our theorem is also applicable to the strong explicit cheat formulation. For this variant of covert security, our theorem can

also be extended to consider an offline hybrid functionality that takes inputs from all parties, in contrast to the definition of the offline function we specified above.

Proof sketch. We present a proof sketch together with the simulator here and defer the full indistinguishability proof to the full version of the paper [FHKS23].

On a high level, we prove our theorem by constructing a simulator \mathcal{S} for the protocol π_{on} in the $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ -hybrid world. In the construction, we exploit the fact that π_{on} is covertly secure in the $\mathcal{F}_{\text{off}}^1$ -hybrid world with deterrence factor ϵ_{on} , which means that a simulator \mathcal{S}_1 for the $\mathcal{F}_{\text{on}}^{\epsilon_{\text{on}}}$ -ideal world exists. Next, we state the full simulator description.

0. Initially, \mathcal{S} calls \mathcal{S}_1 to obtain a random tape used for the execution of Adv .
1. In the first step, \mathcal{S} receives the messages sent from Adv to $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$, i.e., a set of inputs for the corrupted parties $\{x_i^{\text{off}}\}_{i \in \mathcal{I}}$ together with additional input from the adversary $m \in \{\perp, \text{abort}, (\text{cheat}_i, \epsilon_i)\}$, where $i \in \mathcal{I}$ and $\epsilon_i \geq \epsilon'_{\text{off}}$. \mathcal{S} distinguishes the following cases:
- (a) If $m \in \{\perp, \text{abort}\}$, \mathcal{S} sends $\{x_i^{\text{off}}\}_{i \in \mathcal{I}}$ and m to \mathcal{S}_1 and continues the execution exactly as \mathcal{S}_1 . The latter is done by forwarding all messages received from \mathcal{S}_1 to Adv or $\mathcal{F}_{\text{on}}^{\epsilon'_{\text{on}}}$ and vice versa.
 - (b) If $m = (\text{cheat}_\ell, \epsilon_\ell)$ for some $\ell \in \mathcal{I}$, \mathcal{S} samples dummy inputs $\{\hat{x}_i^{\text{on}}\}_{i \in \mathcal{I}}$ for the corrupted parties, sends $\{\hat{x}_i^{\text{on}}\}_{i \in \mathcal{I}}$ together with $(\text{cheat}_\ell, \epsilon_\ell)$ to $\mathcal{F}_{\text{on}}^{\epsilon'_{\text{on}}}$ and distinguishes the following cases:
 - i. If $\mathcal{F}_{\text{on}}^{\epsilon'_{\text{on}}}$ replies $(\text{corrupted}_i, \{\hat{y}_i^{\text{on}}\}_{i \in \mathcal{I}})$, \mathcal{S} computes the probabilistic function $f_{\text{off}} : (\{\perp\}_{i \notin \mathcal{I}}, \{x_i^{\text{off}}\}_{i \in \mathcal{I}}) \rightarrow (\hat{y}_1^{\text{off}}, \dots, \hat{y}_n^{\text{off}})$ using fresh randomness, sends $(\text{corrupted}_i, \{\hat{y}_i^{\text{off}}\}_{i \in \mathcal{I}})$ to Adv and returns whatever Adv returns.
 - ii. Otherwise, if $\mathcal{F}_{\text{on}}^{\epsilon'_{\text{on}}}$ replies $(\text{undetected}, \{x_j^{\text{on}}\}_{j \notin \mathcal{I}})$, \mathcal{S} sends undetected to Adv and gets back the value y defined as follows:
 - If $y \in \{\text{abort}, \text{corrupted}_\ell\}$ for $\ell \in \mathcal{I}$, \mathcal{S} sends y to $\mathcal{F}_{\text{on}}^{\epsilon'_{\text{on}}}$ and returns whatever Adv returns.
 - If $y = \{y_j^{\text{off}}\}_{j \notin \mathcal{I}}$ with $y_j^{\text{off}} \in \{0, 1\}^*$ for $j \notin \mathcal{I}$, \mathcal{S} interacts with Adv to simulate the rest of the protocol. To this end, \mathcal{S} takes x_j^{on} as the input of the honest party P_j and y_j^{off} as P_j 's output of the offline phase for every $j \notin \mathcal{I}$. When the protocol ends with an honest party's output y_j^{on} for $j \notin \mathcal{I}$, \mathcal{S} forwards these outputs to $\mathcal{F}_{\text{on}}^{\epsilon'_{\text{on}}}$ and returns whatever Adv returns. Note that y_j^{on} can also be abort or corrupted_ℓ for $\ell \in \mathcal{I}$.

Recall that due to first restriction on π_{on} , the call to the hybrid functionality $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ is the first message sent in the protocol. Via this message, the adversary Adv decides if it sends cheat to the hybrid functionality or not. Since this message is the first one, the cheat decision depends only on the adversary's code and its random tape. The cheat decision is equally distributed in the hybrid and the

ideal world, as it depends only on the random tape and input of Adv which is the same in the ideal world and in the hybrid world.

In the ideal world, the hybrid functionality is simulated by the simulator \mathcal{S} and hence \mathcal{S} gets the message of Adv . Depending on Adv 's decision to cheat, \mathcal{S} distinguishes between two cases.

On the one hand, in case the adversary *does not cheat*, \mathcal{S} internally runs \mathcal{S}_1 for the remaining simulation. Since the case of no cheating might also appear in the $\mathcal{F}_{\text{off}}^1$ -hybrid world, \mathcal{S}_1 is able to produce an indistinguishable view in the ideal world. We formally show via a reduction to the assumption that π_{on} is covertly secure in the $\mathcal{F}_{\text{off}}^1$ -hybrid world that the views are indistinguishable in this case.

On the other hand, in case the adversary *tries to cheat*, \mathcal{S} cannot use \mathcal{S}_1 . This is due to the fact that the scenario of undetected cheating can occur in the $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ -hybrid world, while it cannot happen in the $\mathcal{F}_{\text{off}}^1$ -hybrid world. Thus, \mathcal{S} needs to be able to simulate undetected cheating which is not required from \mathcal{S}_1 . Instead of using \mathcal{S}_1 , \mathcal{S} simulates the case of cheating on its own. To this end, \mathcal{S} asks the ideal functionality whether or not cheating is detected. If cheating is detected, the remaining simulation is mostly straightforward. One subtlety we like to highlight here is that \mathcal{S} needs to provide the output values of the corrupted parties of $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ to Adv . \mathcal{S} obtains these values by computing the offline function f_{off} . Since this function is independent of the inputs of honest parties, \mathcal{S} is indeed able to compute values that are indistinguishable to the values in the hybrid world execution.

If cheating is undetected, \mathcal{S} needs to simulate the remaining steps of π_{on} . Note that if cheating is undetected, \mathcal{S} obtains the inputs of the honest parties from the ideal functionality. Moreover, the adversary provides to \mathcal{S} the potentially corrupted output values of the hybrid functionality for the honest parties. Now, \mathcal{S} knows all information to act exactly like honest parties do in the hybrid world execution and therefore the resulting view is indistinguishable as well.

We finally give the idea about the deterrence factor of π_{on} in the $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ -hybrid world. We know that cheating during all steps after the call to the hybrid functionality is detected with probability ϵ_{on} . This is due to the fact that π_{on} is covertly secure with deterrence factor ϵ_{on} in the $\mathcal{F}_{\text{off}}^1$ -hybrid world. Now, any cheat attempt in the hybrid functionality is detected only with probability ϵ'_{off} . Since the adversary can decide when he wants to cheat, the detection probability of π_{on} in the $\mathcal{F}_{\text{off}}^{\epsilon'_{\text{off}}}$ -hybrid world is $\epsilon'_{\text{on}} = \min(\epsilon_{\text{on}}, \epsilon'_{\text{off}})$.

4 Covert Online Protocol

In this section, we demonstrate the applicability of our new paradigm to achieve covert security. To this end, we construct a covertly secure online phase for the TinyOT protocol [NNOB12]. We refer to Section 1.3 for the intuition and high-level idea of TinyOT. Here, we present the exact specification of our covertly secure online protocol. We present our protocol in a hybrid world where the

offline phase is modeled via a hybrid functionality and show its covert security under the intermediate explicit cheat formulation (IECF) (cf. Definition 1) in the random oracle model.

In the following, we first present the notation we use to describe our protocol. Then, we state the building blocks of our protocol, especially, an ideal commitment functionality and the offline functionality, which are both used as hybrid functionalities. Next, we present the exact specification of our two-party online protocol and afterwards prove its security.

We remark that we focus on the two-party setting, since this setting is sufficient to show applicability and the benefit of our paradigm. Nevertheless, we believe our protocol can easily be extended to the multi-party case following the multi-party extensions of TinyOT ([LOS14, BLN⁺21, FKOS15, WRK17b]).

Notation. We use the following notation to describe secret shared and authenticated values. This notation follows the common approach in the research field [NNOB12, DPSZ12, WRK17a, WRK17b]. For covert security parameter t , both parties have a global key, $\Delta_{\mathcal{A}}$ resp. $\Delta_{\mathcal{B}}$, which are bit strings of length t . A bit x is authenticated to a party \mathcal{A} by having the other party \mathcal{B} hold a random t -bit key, $K[x]$, and having \mathcal{A} hold the bit x and a t -bit MAC $M[x] = K[x] \oplus x \cdot \Delta_{\mathcal{B}}$. We denote an authenticated bit x known to \mathcal{A} as $\langle x \rangle^{\mathcal{A}}$ which corresponds to the tuple $(x, K[x], M[x])$ in which x and $M[x]$ is known by \mathcal{A} and $K[x]$ by \mathcal{B} . A public constant c can be authenticated to \mathcal{A} non-interactively by defining $\langle c \rangle^{\mathcal{A}} := (c, c \cdot \Delta_b, 0^{\kappa})$. Authenticated bits known to \mathcal{B} are authenticated and denoted symmetrically.

A bit z is secret shared by having \mathcal{A} hold a value x and \mathcal{B} hold a value y such that $z = x \oplus y$. The secret shared bit is authenticated by authenticating the individual shares of \mathcal{A} and \mathcal{B} , i.e., by using $\langle x \rangle^{\mathcal{A}}$ and $\langle y \rangle^{\mathcal{B}}$. We denote the authenticated secret sharing $(\langle x \rangle^{\mathcal{A}}, \langle y \rangle^{\mathcal{B}}) = (x, K[x], M[x], y, K[y], M[y])$ by $\langle z \rangle$ or $\langle x|y \rangle$.

Observe that this kind of authenticated secret sharing allows linear operations, i.e., addition of two secret shared values as well as addition and multiplication of a secret shared value with a public constant. In order to calculate $\langle \gamma \rangle := \langle \alpha \rangle \oplus \langle \beta \rangle$ with $\langle \alpha \rangle = \langle a_{\mathcal{A}}|a_{\mathcal{B}} \rangle$, $\langle \beta \rangle = \langle b_{\mathcal{A}}|b_{\mathcal{B}} \rangle$, parties compute the authenticated share of γ of \mathcal{A} as $\langle c_{\mathcal{A}} \rangle^{\mathcal{A}} := (a_{\mathcal{A}} \oplus b_{\mathcal{A}}, K[a_{\mathcal{A}}] \oplus K[b_{\mathcal{A}}], M[a_{\mathcal{A}}] \oplus M[b_{\mathcal{A}}])$. The authenticated share of γ of \mathcal{B} , $\langle c_{\mathcal{B}} \rangle^{\mathcal{B}}$, is calculated symmetrically. It follows that $\langle \gamma \rangle = \langle c_{\mathcal{A}}|c_{\mathcal{B}} \rangle$ is an authenticated sharing of $\alpha \oplus \beta$. In order to calculate $\langle \gamma \rangle := \langle \alpha \rangle \oplus \beta$ for a public constant β and α defined as above, parties first create authenticated constants bits $\langle \beta \rangle^{\mathcal{A}}$ and $\langle 0 \rangle^{\mathcal{B}}$ and define $\langle \beta \rangle := \langle \beta|0 \rangle$. In order to calculate $\langle \gamma \rangle := \langle \alpha \rangle \cdot \beta$ for a public constant β and α defined as above, parties set $\langle \gamma \rangle := \langle \alpha \rangle$ if $b = 1$ and $\langle \gamma \rangle := \langle 0|0 \rangle$ if $b = 0$.

Finally, we use the notation $[n]$ to denote the set $\{1, \dots, n\}$. We consider any sets to be ordered, e.g., $\{x_i\}_{i \in [n]} := [x_1, x_2, \dots, x_n]$, and for a set of indices $\mathcal{I} = \{x_i\}_{i \in [n]}$ we denote the i -th element of \mathcal{I} as $\mathcal{I}[i]$. Note, that $M[x]$ always denotes a MAC for bit x and we only denote the i -th element for sets of indices which we denote by \mathcal{I} .

Ideal commitments. The protocol uses an hybrid commitment functionality $\mathcal{F}_{\text{Commit}}$ that is specified as follows:

Functionality $\mathcal{F}_{\text{Commit}}$: Commitments

The functionality interacts with two parties, \mathcal{A} and \mathcal{B} .

- Upon receiving (Commit, x_P) from party $P \in \{\mathcal{A}, \mathcal{B}\}$, check if Commit was not received before from P . If the check holds, store x_P and send $(\text{Committed}, P)$ to party $\bar{P} \in \{\mathcal{A}, \mathcal{B}\} \setminus P$.
- Upon receiving (Open) from party $P \in \{\mathcal{A}, \mathcal{B}\}$, check if Commit was received before from P . If the check holds, send (Open, P, x_P) to party $\bar{P} \in \{\mathcal{A}, \mathcal{B}\} \setminus P$.

Offline functionality. The online protocol uses an hybrid offline functionality $\mathcal{F}_{f_{\text{off}}}^\epsilon$ to provide authenticated bits and authenticated triples. Function f_{off} is defined as follows.

Functionality f_{off} : Precomputation

The function receives inputs by two parties, \mathcal{A} and \mathcal{B} . W.l.o.g., we assume that if any party is corrupted it is \mathcal{A} . The function is parametrized with a number of authenticated bits, n_1 , a number of authenticated triples n_2 and the deterrence parameter t .

Inputs: \mathcal{A} provides either input ok or $(\Delta_{\mathcal{A}}, \{r_i, K[s_i], M[r_i]\}_{i \in [n_1 + 3 \cdot n_2]})$ where $\Delta_{\mathcal{A}}, K[\cdot], M[\cdot]$ are t -bit strings and r_i is a bit for $i \in [n_1 + 3 \cdot n_2]$. An honest \mathcal{A} will always provide input ok . \mathcal{B} provides input ok .

Computation: The function calculates authenticated bits and authenticated shared triples as follows:

- Sample $\Delta_{\mathcal{B}} \in_R \{0, 1\}^t$. Do the same for $\Delta_{\mathcal{A}}$ if not provided as input.
- For each $i \in [n_1 + 3 \cdot n_2]$, sample $s_i \in_R \{0, 1\}$. If not provided as input, sample $r_i \in_R \{0, 1\}$ and $K[s_i], M[r_i] \in_R \{0, 1\}^t$. Set $K[r_i] := M[r_i] \oplus r_i \cdot \Delta_{\mathcal{B}}$ and $M[s_i] := K[s_i] \oplus s_i \cdot \Delta_{\mathcal{A}}$. Define $\langle r_i \rangle^{\mathcal{A}} := (r_i, K[r_i], M[r_i])$ and $\langle s_i \rangle^{\mathcal{B}} = (s_i, K[s_i], M[s_i])$.
- For each $i \in [n_2]$, set $j = n_1 + 3 \cdot i$ and define $x := r_j \oplus (r_{j-1} \oplus s_{j-1}) \cdot (r_{j-2} \oplus s_{j-2})$, $K[x] := K[s_j]$, and $M[x] := K[x] \oplus x \cdot \Delta_{\mathcal{A}}$ and $\langle x \rangle^{\mathcal{B}} := (x, K[x], M[x])$. Then, define the multiplication triple $\langle \alpha_i \rangle := \langle r_{j-2} | s_{j-2} \rangle$, $\langle \beta_i \rangle := \langle r_{j-1} | s_{j-1} \rangle$, and $\langle \gamma_i \rangle := \langle r_j | x \rangle$.

Output: Output global keys $(\Delta_{\mathcal{A}}, \Delta_{\mathcal{B}})$, authenticated bits $\{\langle r_i \rangle^{\mathcal{A}}, \langle s_i \rangle^{\mathcal{B}}\}_{i \in [n_1]}$, and authenticated shared triples $\{\langle \alpha_i \rangle, \langle \beta_i \rangle, \langle \gamma_i \rangle\}_{i \in [n_2]}$, and assign \mathcal{A} and \mathcal{B} their respective shares, keys and macs.

We present a protocol instantiating $\mathcal{F}_{f_{\text{off}}}^\epsilon$ in the full version of the paper [FHKS23].

Online protocol. The online protocol works in four steps. First, the parties obtain authenticated bits and triples from the hybrid offline functionality. Second, the parties secret share their inputs and use authenticated bits to obtain au-

authenticated shares of the inputs wires of the circuit. Third, the parties evaluate the boolean circuit on the authenticated values. While XOR-gates are computed locally, AND-gates require communication between the parties and the consumption of a precomputed authenticated triple for each gate. Finally, in the output phase each party verifies the MACs on the computed values to check for correct behavior of the other party. If no cheating was detected, the parties exchange their shares on the output wires to recompute the actual outputs.

We modified the original TinyOT online phase in two aspects. First, the original TinyOT protocol uses one-sided authenticated precomputation data, e.g., one-sided authenticated triples where the triple is not secret shared but known to one party. In contrast, we focus on a simplification [WRK17a] where the authenticated triples are secret shared among all parties. This allows us to use a single two-sided authenticated triple for each AND gate instead of two one-sided authenticated triples with additional data. Second, we integrate commitments in the output phase. In detail, the parties first commit on their shares for the output wires together with the corresponding MACs and only afterwards reveal the committed values. By using commitments, the adversary needs to decide first if it wants to cheat and only afterwards it learns the output. However, since the adversary can commit on incorrect values, it still can learn its output even if the honest parties detect its cheating afterwards. We show the security of this protocol under the IECF of covert security.

To prevent the adversary from inserting incorrect values into the commitment, the generation of the commitments can be part of the circuit evaluation. By checking the correct behavior of the entire evaluation, honest parties detect cheating with the inputs to the commitments with a fixed probability. This way, we can achieve the strong explicit cheat formulation (SECF). Since computing the commitments as part of the circuit reduces the efficiency, we opted for the less expensive protocol.

Protocol Π_{on} : TinyOT-style online protocol

The protocol is executed between parties \mathcal{A} and \mathcal{B} and uses of a hash function H (modeled as non-programmable random oracle), the hybrid commitment functionality $\mathcal{F}_{\text{Commit}}$, and the hybrid covert functionality $\mathcal{F}_{f_{\text{off}}}^1$, in the following denoted as \mathcal{F}_{off} . f_{off} is instantiated with the same public parameters as the protocol. When denoting a particular party with P , we denote the respective other party with \bar{P} .

Public parameters: The deterrence parameter t and the number of input bits and output bits per party n_1 . A function $f(\{x_{(i,\mathcal{A})}\}_{i \in [n_1]}, \{x_{(i,\mathcal{B})}\}_{i \in [n_1]}) = (\{z_{(i,\mathcal{A})}\}_{i \in [n_1]}, \{z_{(i,\mathcal{B})}\}_{i \in [n_1]})$ with $x_{(*,\mathcal{A})}, x_{(*,\mathcal{B})}, z_{(*,\mathcal{A})}, z_{(*,\mathcal{B})} \in \{0, 1\}$ and a boolean circuit \mathcal{C} computing f with n_2 AND gates. $\{z_{(i,\mathcal{A})}\}_{i \in [n_1]}$ resp. $\{z_{(i,\mathcal{B})}\}_{i \in [n_1]}$ is the output of \mathcal{A} resp. \mathcal{B} . The set of indices of input wires resp. output wires of each party $P \in \{\mathcal{A}, \mathcal{B}\}$ is denoted by $\mathcal{I}_P^{\text{in}}$ resp. $\mathcal{I}_P^{\text{out}}$. Without loss of generality, we assume that the wire values are ordered in topological order.

Inputs: \mathcal{A} has input bits $\{x_{(i,\mathcal{A})}\}_{i \in [n_1]}$ and \mathcal{B} has input bits $\{x_{(i,\mathcal{B})}\}_{i \in [n_1]}$.

Pre-computation phase:

1. Each party $P \in \{\mathcal{A}, \mathcal{B}\}$ defines ordered sets $\mathcal{M}_P^P := \emptyset$, $\mathcal{M}_{\bar{P}}^P := \emptyset$, sends (ok) to \mathcal{F}_{off} and receives its shares of $(\{\langle r_{(i,\mathcal{A})} \rangle^{\mathcal{A}}, \langle r_{(i,\mathcal{B})} \rangle^{\mathcal{B}}\}_{i \in [n_1]}, \{\langle \alpha_j \rangle, \langle \beta_j \rangle, \langle \gamma_j \rangle\}_{j \in [n_2]})$. If \mathcal{F}_{off} , returns $m \in \{\text{abort}, \text{corrupted}_{\bar{P}}\}$, P outputs m and aborts.

Input phase:

2. For each $i \in [n_1]$, each party $P \in \{\mathcal{A}, \mathcal{B}\}$ sends $d_{(i,P)} := x_{(i,P)} \oplus r_{(i,P)}$. Then, the parties define $\langle x_{(i,\mathcal{A})} \rangle := \langle r_{(i,\mathcal{A})} | 0 \rangle \oplus d_{(i,\mathcal{A})}$ and $\langle x_{(i,\mathcal{B})} \rangle := \langle 0 | r_{(i,\mathcal{B})} \rangle \oplus d_{(i,\mathcal{B})}$. For each party $P \in \{\mathcal{A}, \mathcal{B}\}$ and each $j \in [n_1]$ with $i := \mathcal{I}_P^n[j]$, the parties assign $\langle x_{(j,P)} \rangle$ to $\langle w_i \rangle$.

Circuit evaluation phase:

3. Repeat till all wire values are assigned. Let j be the smallest index of an unassigned wire. Let l and r be the indices of the left resp. right input wire of the gate computing w_j . Dependent on the gate type, $\langle w_j \rangle$ is calculated as follows:

- **XOR-Gate:** $\langle w_j \rangle := \langle w_l \rangle \oplus \langle w_r \rangle$
- **AND-Gate:** For the i -th AND gate, the parties define $(\langle \alpha \rangle, \langle \beta \rangle, \langle \gamma \rangle) := (\langle \alpha_i \rangle, \langle \beta_i \rangle, \langle \gamma_i \rangle)$, calculate $\langle e \rangle = \langle e^{\mathcal{A}} | e^{\mathcal{B}} \rangle := \langle \alpha \rangle \oplus \langle w_l \rangle$ and $\langle d \rangle = \langle d^{\mathcal{A}} | d^{\mathcal{B}} \rangle := \langle \beta \rangle \oplus \langle w_r \rangle$, open e and d by publishing $e^{\mathcal{A}}, e^{\mathcal{B}}, d^{\mathcal{A}}, d^{\mathcal{B}}$ respectively, and compute $\langle w_j \rangle := \langle \gamma \rangle \oplus e \cdot \langle w_r \rangle \oplus d \cdot \langle w_l \rangle \oplus e \cdot d$. Further, each party $P \in \{\mathcal{A}, \mathcal{B}\}$ appends $(M[e^P], M[d^P])$ to \mathcal{M}_P^P and $((K[e^{\bar{P}}] \oplus e^{\bar{P}} \cdot \Delta_P), (K[d^{\bar{P}}] \oplus d^{\bar{P}} \cdot \Delta_P))$ to $\mathcal{M}_{\bar{P}}^P$.

Output phase:

4. Party $P \in \{\mathcal{A}, \mathcal{B}\}$ computes $\mathcal{M}_{(P,P)}^1 := H(\mathcal{M}_P^P)$ and $\mathcal{M}_{(P,\bar{P})}^1 = H(\mathcal{M}_{\bar{P}}^P)$ and sends $\mathcal{M}_{(P,P)}^1$.
5. Each party $P \in \{\mathcal{A}, \mathcal{B}\}$, upon receiving $\mathcal{M}_{(\bar{P},\bar{P})}^1$, verifies that $\mathcal{M}_{(\bar{P},\bar{P})}^1 = \mathcal{M}_{(P,\bar{P})}^1$. If not, P outputs $\text{corrupted}_{\bar{P}}$ and aborts. Otherwise, P computes $\mathcal{M}_{(P,P)}^2 := H(\{M[w_i^P]\}_{i \in \mathcal{I}_P^{\text{out}}})$, and sends (Commit, $(\{w_i^P\}_{i \in \mathcal{I}_P^{\text{out}}}, \mathcal{M}_{(P,P)}^2)$) to $\mathcal{F}_{\text{Commit}}$.
6. Upon receiving, (Committed, \bar{P}) from $\mathcal{F}_{\text{Commit}}$, P sends (Open) to $\mathcal{F}_{\text{Commit}}$.
7. Each party $P \in \{\mathcal{A}, \mathcal{B}\}$, upon receiving (Opened, \bar{P} , $(\{w_i^{\bar{P}}\}_{i \in \mathcal{I}_{\bar{P}}^{\text{out}}}, \mathcal{M}_{(\bar{P},\bar{P})}^2)$) from $\mathcal{F}_{\text{Commit}}$, re-defines $\mathcal{M}_{\bar{P}}^P := \{K[w_i^{\bar{P}}] \oplus w_i^{\bar{P}} \cdot \Delta_P\}_{i \in \mathcal{I}_{\bar{P}}^{\text{out}}}$ and verifies that $\mathcal{M}_{(\bar{P},\bar{P})}^2 = H(\mathcal{M}_{\bar{P}}^P)$. If not, P outputs $\text{corrupted}_{\bar{P}}$ and aborts. Otherwise, P outputs $\{w_i^P \oplus w_i^{\bar{P}}\}_{i \in \mathcal{I}_P^{\text{out}}}$.

Handle aborts:

8. If a party P does not receive a timely message before executing Step 6, it outputs abort and aborts. If a party P does not receive a timely message after having executed Step 6, it outputs $\text{corrupted}_{\bar{P}}$ and aborts.

Security. Intuitively, successful cheating in the context of the online protocol is equivalent to correctly guessing the global key of the other party. Let us assume \mathcal{A} is corrupted. It is evident that \mathcal{A} can only behave maliciously by flipping the bits sent during the evaluation phase and the output phase – flipping a bit during the input phase is not considered cheating as the adversary, \mathcal{A} , is allowed to pick its input arbitrarily. For each of those bits, there is a MAC check incorporated into the protocol. Hence, \mathcal{A} needs to guess the correct MACs for the flipped bits (\mathcal{A} knows the ones of the unflipped bits) in order to cheat successfully. As a MAC $M[b^A]$ for a bit b^A known to \mathcal{A} is defined as $K[b^A] \oplus b^A \cdot \Delta_{\mathcal{B}}$, a MAC $\widetilde{M}[\tilde{b}^A]$ of a flipped bit \tilde{b}^A is correct iff $\widetilde{M}[\tilde{b}^A] = M[b^A] \oplus \Delta_{\mathcal{B}} = K[b^A] \oplus (b^A \oplus 1) \cdot \Delta_{\mathcal{B}}$. It follows that \mathcal{A} has to guess the global key of \mathcal{B} and apply it to the MACs of all flipped bits in order to cheat successfully. As the global key has t bits, the chance of guessing the correct global key is $\frac{1}{2^t}$. It follows that the deterrence factor ϵ equals $1 - \frac{1}{2^t}$. More formally, we state the following theorem and prove its correctness in the full version of the paper [FHKS23]:

Theorem 2. *Let H be a (non-programmable) random oracle, $t \in \mathbb{N}$, and $\epsilon = 1 - \frac{1}{2^t}$. Then, protocol Π_{on} securely implements \mathcal{F}_f^ϵ (i.e., constitutes a covertly secure protocol with deterrence factor ϵ) in the presence of a rushing adversary according to the intermediate explicit cheat formulation as defined in Definition 1 in the $(\mathcal{F}_{\text{off}}, \mathcal{F}_{\text{Commit}})$ -hybrid world.*

On the usage of random oracles. As explained above, successful cheating is equivalent to guessing the global key of the other party. However, a malicious party can also cheat inconsistently, i.e., it guesses different global keys for the flipped bits, or even provide incorrect MACs for unflipped bits. In this case, the adversary has no chance of cheating successfully, which needs to be detected by the simulator. As the simulator only receives a hash of all MACs, it needs some trapdoor to learn the hashed MACs and check for consistency. To provide such a trapdoor, we model the hash function as a random oracle. The requirement of a random oracle can be removed if the parties send all MACs in clear instead of hashing them first. However, this increases the communication complexity.

Another alternative is to bound the deterrence parameter t such that the simulator can try out all consistent ways to compute the MACs of flipped bits, i.e., each possible value for the guessed global key, hash those and compare them to the received hash. In this case, it is sufficient to require collision resistance of the hash function. As the number of possible values for the global key grows exponentially with the deterrence parameter t , i.e., 2^t , this approach is only viable if we bound t . Nevertheless, the probability of successful cheating also declines exponentially with t , i.e., $\frac{1}{2^t}$. Hence, for small values of t , the simulator runs in reasonable time.

5 Evaluation

In Section 4, we showed the application of our new paradigm to achieve covert security on the example of the TinyOT online phase. By shortening the MAC

length in the online phase, we also reduced the amount of precomputation required from the offline phase. In order to quantify the efficiency gain that can be achieved by generating shorter MACs, we compare the communication complexity of a covert offline phase generating authenticated bits and triples with short MACs to the covert offline phase generating bits and triples with long MACs.

The offline protocol. To the best of our knowledge, there is no explicit covert protocol for the precomputation of TinyOT-style protocols. Therefore, we rely on generic transformations from semi-honest to covert security based on the cut-and-choose paradigm, similar to the transformations proposed by [DOS20, FHKS21, SSS22]. However, semi-honest precomputation protocols do not consider authentication of bits and triples, since semi-honest online protocols do not need authentication. Hence, it is necessary to first extend the semi-honest protocol to generate MACs, and then, apply the generic transformation. We first specify a semi-honest protocol to generate authenticated bits and triples as well as the covert protocol that can be derived via the cut-and-choose approach. Both protocols are presented in the full version of the paper [FHKS23]. Then, we take the resulting covert protocol to evaluate the communication complexity for different MAC lengths.

ϵ	# triples	λ -bit MACs (state-of-the-art)	Short MACs (our approach)	Improvement
$\frac{1}{2}$	10 K	531	333	37,19%
	100 K	5 211	3 258	37,47%
	1 M	52 011	32 508	37,50%
	1 B	52 000 011	32 500 008	37,50%
$\frac{3}{4}$	10 K	1 062	677	36,24%
	100 K	10 422	6 617	36,51%
	1 M	104 022	66 017	36,54%
	1 B	104 000 022	66 000 017	36,54%
$\frac{7}{8}$	10 K	2 124	1 374	35,29%
	100 K	20 844	13 434	35,55%
	1 M	208 044	134 034	35,57%
	1 B	208 000 044	134 000 034	35,58%

Table 1: Concrete communication complexity of the covert offline phase generating the precomputation required for a maliciously secure TinyOT online phase (as applied by state-of-the-art) and a covertly secure TinyOT online phase (our approach). As the offline phase is covertly secure, the overall protocol’s security level is the same in both approaches. Communication is reported in kB per party.

Evaluation results. The communication complexity of each party is determined as follows. Let κ be the computational security parameter, λ be the statistical security parameter, t be the cut-and-choose parameter (which results in a deterrence factor $\epsilon = 1 - \frac{1}{t}$), M be the length of the generated MACs, n_1 be the number of authenticated bits required per party, n_2 be the number of

authenticated triples, C_{OT} be the communication complexity of one party for performing κ base oblivious transfers with κ -bit strings twice, once as receiver and once as sender, C_{Commit} be the size of a commitment and C_{Open} be the size of an opening to a κ -bit seed. Then, each party needs to send C bits with C equal to

$$(t + 1) \cdot C_{\text{Commit}} + t \cdot (C_{\text{OT}} + C_{\text{Open}} + n_2 \cdot (3 + \kappa - 1) + (n_1 + 2 \cdot n_2) \cdot (M - 1))$$

In our approach, M is defined such that $t = 2^M$. In the classical approach with a maliciously secure online phase M is fixed to equal λ . This yields an absolute efficiency gain of G bits with G equal to

$$t \cdot (n_1 + 2 \cdot n_2) \cdot (\lambda - M)$$

In the following, we set $\kappa = 128$, $\lambda = 40$, $C_{\text{OT}} = (2 + \kappa) \cdot 256$ according to [MRR21], $C_{\text{Commit}} = 256$ and $C_{\text{Open}} = 2 \cdot \kappa$ according to a hash-based commitment scheme. Further, we fix $n_1 = 256$. This yields the communication complexity depicted in Table 1. For deterrence factors up to $\frac{7}{8}$, our approach reduces the communication per party by at least 35%. As a reduction of the security of the online phase to the level of the offline phase does not affect the overall protocol’s security, as shown in Section 3.2, this efficiency improvement is for free.

Acknowledgments

The first, third, and fourth authors were supported by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 – 236615297 (CROSSING Project S7)*, and by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*. The second author was supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister’s Office, and by ISF grant No. 1316/18.

Appendix

A Discussion of Constraints on Online Protocol

In this section, we discuss the constraints on the online protocol used in our theorem. These constraints emerged from technical issues and it is unclear how to prove our deterrence replacement theorem in a more generic setting. Recall

that in our proof \mathcal{S} uses the simulator \mathcal{S}_1 which exists since π_{on} is covertly secure in the $\mathcal{F}_{\text{off}}^1$ -hybrid world.

First, the hybrid functionality \mathcal{F}_{off} needs to be called directly at the beginning. This enables the simulator \mathcal{S} to react to the adversary's cheating decision in the offline phase, i.e., its input to \mathcal{F}_{off} , right at the start of the simulation. More specifically, \mathcal{S} uses the black-box simulator \mathcal{S}_1 in case the adversary does not cheat and simulates on its own in case there is a cheating attempt. If there would be protocol interactions before the call to \mathcal{F}_{off} , \mathcal{S} would have to decide whether it simulates this interactions itself or via \mathcal{S}_1 . This means that the adversary's input to \mathcal{F}_{off} could require \mathcal{S} to change its decision, e.g., require \mathcal{S} to simulate the following steps itself while \mathcal{S} initially used \mathcal{S}_1 for the earlier steps. This leads to a problem as \mathcal{S} uses \mathcal{S}_1 in a black-box way, and hence, can only use it for all or none of the protocol steps. Rewinding does not solve the problem as a change in the simulation of the steps before the call to \mathcal{F}_{off} can influence the adversary's input to \mathcal{F}_{off} , and hence, \mathcal{S} 's decision to simulate the steps afterwards based on \mathcal{S}_1 or not.

Second, we require that in case \mathcal{F}_{off} outputs `corrupted`, the protocol π_{on} instructs the parties to output `corrupted` as well. This is due to some subtle detail in the security proof. As \mathcal{S}_1 runs in a world, in which cheating in the offline phase is not possible, \mathcal{S}_1 does not know how to deal with undetected cheating. Further, we treat the protocol π_{on} in a black-box way. Due to these facts, the only way for \mathcal{S} to simulate the case of undetected cheating is to follow the actual protocol. To do so in a consistent way, \mathcal{S} has to get the input of the honest parties. Hence, \mathcal{S} has to notify the ideal covert functionality $\mathcal{F}_{\text{on}}^{\epsilon_{\text{on}}}$ about the cheating attempt in the offline phase. In case of detected cheating, $\mathcal{F}_{\text{on}}^{\epsilon_{\text{on}}}$ sends `corrupted` to the honest parties and thus the honest parties output `corrupted` in the ideal world. In order to achieve indistinguishability between the ideal world and the real world, π_{on} needs to instruct the honest parties to output `corrupted` in the real world, too.

Finally, we emphasize that known offline/online protocols (SPDZ [DPSZ12], TinyOT [NNOB12], authenticated garbling [WRK17a, WRK17b]) either directly fulfill the aforementioned requirements or can easily be adapted to do so.

B Comparison of Theorem 1 with [AL07]

Aumann and Lindell [AL07] presented a sequential composition theorem for the (strong) explicit cheat formulation. The theorem shows that a protocol π that is covertly secure in an $(\mathcal{F}_1^{\epsilon_1}, \dots, \mathcal{F}_{p(n)}^{\epsilon_{p(n)}})$ -hybrid world with deterrence factor ϵ_π , i.e., parties have access to a polynomial number of functionalities $\mathcal{F}_1, \dots, \mathcal{F}_{p(n)}$ with deterrence factor $\epsilon_1, \dots, \epsilon_{p(n)}$, respectively, is also covertly secure with deterrence ϵ_π if functionality \mathcal{F}_i is replaced by a protocol π_i that realizes \mathcal{F}_i with deterrence factor ϵ_i for $i \in \{1, \dots, p(n)\}$. This theorem allows to analyze the security of a protocol in a hybrid model and replace the hybrid functionalities with subprotocols afterwards. Aumann and Lindell already noted that the computation of the deterrence factor ϵ_π needs to take all the deterrence factors of

the subprotocols into account. However, the theorem does not make any statement about how the individual deterrence factors influence the deterrence factor of the overall protocol and neither analyzes the effect of changing some of the deterrence factors ϵ_i .

Our theorem takes one step further and addresses the aforementioned drawbacks. In particular, it allows to analyze the security of a protocol in a *simple* hybrid world, in which the hybrid functionality is associated with deterrence factor 1. As there is no successful cheating in the hybrid functionality, a proof in this hybrid world is expected to be much simpler. The same holds for the calculation of the overall deterrence factor. Once having proven a protocol to be secure in the simple hybrid world, our theorem allows to derive the security and the deterrence factor of the same protocol in the hybrid world, in which the offline phase is associated with some smaller deterrence factor, $\epsilon' \in [0, 1]$.

References

- ABL⁺18. David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 2018.
- AL07. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, 2007.
- AO12. Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In *ASIACRYPT*, 2012.
- BCG⁺19. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO*, 2019.
- BCG⁺20a. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density LPN. In *FOCS*, 2020.
- BCG⁺20b. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *CRYPTO*, 2020.
- BCS19. Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using topgear in overdrive: A more efficient zkpk for SPDZ. In *SAC*, 2019.
- BLN⁺21. Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High-performance multi-party computation for binary circuits based on oblivious transfer. *J. Cryptol.*, 34(3):34, 2021.
- Can00. Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1), 2000.
- CDE⁺18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SpdF_{2^k} : Efficient MPC mod 2^k for dishonest majority. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, 2018.

- CKR⁺20. Hao Chen, Miran Kim, Ilya P. Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. Maliciously secure matrix multiplication with applications to private deep learning. In *ASIACRYPT*, 2020.
- DILO22. Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Authenticated garbling from simple correlations. In *CRYPTO*, 2022.
- DKL⁺13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, 2013.
- DNNR17. Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The tinytable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *CRYPTO*, 2017.
- DOS20. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. In *CRYPTO*, 2020.
- DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multi-party computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.
- DZ13. Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, 2013.
- FHKS21. Sebastian Faust, Carmit Hazay, David Kretzler, and Benjamin Schlosser. Generic compiler for publicly verifiable covert multi-party computation. In *EUROCRYPT*, 2021.
- FHKS23. Sebastian Faust, Carmit Hazay, David Kretzler, and Benjamin Schlosser. Putting the online phase on a diet: Covert security from short macs. Cryptology ePrint Archive, Paper 2023/052, 2023. <https://eprint.iacr.org/2023/052>.
- FKOS15. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *ASIACRYPT*, 2015.
- HOSS18. Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, tinykeys for tinyot). In *ASIACRYPT*, 2018.
- IOZ14. Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO*, 2014.
- KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.
- KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, 2018.
- KRRW18. Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In *CRYPTO*, 2018.
- KVH⁺21. Brian Knott, Shobha Venkataraman, Awni Y. Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *NeurIPS*, 2021.
- LOS14. Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In *CRYPTO*, 2014.
- MPC. MPC Alliance. <https://www.mpcalliance.org/>. (Accessed on 10/14/2022).
- MRR21. Ian McQuoid, Mike Rosulek, and Lawrence Roy. Batching base oblivious transfers. In *ASIACRYPT*, 2021.

- NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO*, 2012.
- Ors20. Emanuela Orsini. Efficient, actively secure MPC with a dishonest majority: A survey. In *WAIFI*, 2020.
- SSS22. Peter Scholl, Mark Simkin, and Luisa Siniscalchi. Multiparty computation with covert security and public verifiability. In *ITC*, 2022.
- VSG⁺19. Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *EuroSys*, 2019.
- WRK17a. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, 2017.
- WRK17b. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.
- YWZ20. Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In *CCS*, 2020.
- Zen. ZenGo - crypto wallet app. <https://zengo.com/>. (Accessed on 10/14/2022).

D. Statement-Oblivious Threshold Witness Encryption

This chapter corresponds the following publication. The full version is available at [92].

- [91] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser. “Statement-Oblivious Threshold Witness Encryption”. In: *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*. 2023, pp. 17–32. **Part of this thesis.**

Statement-Oblivious Threshold Witness Encryption

Sebastian Faust*, Carmit Hazay†, David Kretzler*, Benjamin Schlosser*

**Technical University of Darmstadt*, Darmstadt, Germany

{first.last}@tu-darmstadt.de

† *Bar-Ilan University*, Ramat Gan, Israel

carmit.hazay@biu.ac.il

Abstract—The notion of witness encryption introduced by Garg et al. (STOC’13) allows to encrypt a message under a statement x from some NP-language \mathcal{L} with associated relation $(x, w) \in \mathcal{R}$, where decryption can be carried out with the corresponding witness w . Unfortunately, known constructions for general-purpose witness encryption rely on strong assumptions, and are mostly of theoretical interest. To address these shortcomings, Goyal et al. (PKC’22) recently introduced a blockchain-based alternative, where a committee decrypts ciphertexts when provided with a valid witness w . Blockchain-based committee solutions have recently gained broad interest to offer security against more powerful adversaries and construct new cryptographic primitives.

We follow this line of work, and propose a new notion of *statement-oblivious* threshold witness encryption. Our new notion offers the functionality of committee-based witness encryption while additionally hiding the statement used for encryption. We present two ways to build statement-oblivious threshold witness encryption, one generic transformation based on anonymous threshold identity-based encryption (A-TIBE) and one direct construction based on bilinear maps. Due to the lack of efficient A-TIBE schemes, the former mainly constitutes a feasibility result, while the latter yields a concretely efficient scheme.

Index Terms—Threshold Witness Encryption, Statement Obliviousness, Committee-Based Decryption, Threshold Tag-Based Encryption

I. INTRODUCTION

The notion of *witness encryption* as introduced by Garg et al. [1] allows a party to encrypt a message m under some problem instance x such that the ciphertext can only be decrypted by someone holding a witness w . There are countless applications of witness encryption ranging from public key encryption with fast key generation, attribute-based encryption for general circuits [1], to using it for encrypting a prize for solving an NP-hard puzzle like the millennium problems, or achieving fairness in MPC [2]. More formally, witness encryption is defined for an NP language \mathcal{L} with associated relation $(x, w) \in \mathcal{R}$, where x is the *statement* and w is the corresponding *witness*. Security as defined by Garg et al. [1] states that for any ciphertext that was created for x *not* in the language \mathcal{L} , ciphertexts do not reveal information about the encrypted message. While this notion only deals with statements that are not in the language, Goldwasser et al. [3] introduced the notion of *extractable witness encryption* stating that even for a statement *in* the language, ciphertexts hide the message.

Although great progress has been made over the last years [1], [3]–[7], witness encryption still has limitations. First,

known constructions rely on strong assumptions like multilinear maps [1], [3], [5], [6], indistinguishability obfuscation [4] or cryptographic invariant maps [7], and its constructions are not practically efficient yet. Second, even the stronger notion of extractable witness encryption does not hide the statement for which the ciphertext was created. This rules out interesting applications that require the statement to be private until decryption takes place, as it may disclose sensitive information.

The first shortcoming of state-of-the-art witness encryption can be circumvented via so-called *extractable Witness Encryption on Blockchains* (eWEB) put forward by Goyal et al. [8]. It is based on a blockchain following a recent trend in cryptography, where constructions leverage the power of blockchains, e.g., [2], [9]–[12]. In the context of witness encryption, this results in a shift from relying on strong number theoretic assumptions to relying on an honest quorum of users within a committee. This trend is further fueled by a line of work that presents constructions of how such committees can be obtained in a blockchain setting [10], [13], [14].

In a nutshell, the scheme of [8] works as follows. Parties encrypt a message by secret sharing it to a committee and labeling the shares with a statement x . To decrypt, parties need to send a witness to the committee proving that x is in the language \mathcal{L} and getting the secret shares back. While the construction of Goyal et al. is certainly more efficient than standard general-purpose witness encryption, the downside of their solution is the storage complexity of the committee, which grows linearly with the number of ciphertexts. Improving on the approach of [8], [9] propose as an application for their large-scale non-interactive threshold cryptosystem a solution, in which the decryption committee stores only secret key shares of a labeled threshold encryption scheme. The committee receives ciphertext-witness-pairs and decrypts only if the witness corresponds to the statement encoded as the label of the ciphertext. This reduces the storage complexity to be only constant. Following [8], [9], Campanelli et al. [10] presents a similar construction called *Blockchain Witness Encryption* (BWE). However, their construction is not practical (e.g., for each encryption a smart contract deployment is required).

In this work, we start with the approach of [9], which we abstractly call *threshold witness encryption*, and address the second shortcoming by a new feature called *statement*

obliviousness, which guarantees that the statement is hidden given the ciphertext. This new feature allows us to extend applications of standard (threshold) witness encryption with an additional privacy property. For instance, we can construct time-lock encryption from witness encryption, as proposed by [6], without leaking the concrete time at which a decryption can happen to third parties, or we can construct a dead-man’s switch, as proposed by [8], without revealing for which person it was created. Moreover, this feature enables a new class of applications that inherently require the privacy property and are not covered by standard witness encryption. As a concrete example, imagine a user wants to buy some shares of a company or some tokens on a Decentralized Finance (DeFi) trading platform, once the price of the asset reaches a certain value; however, without the necessity of having to stay online. Privacy is an important aspect in this scenario, since revealing information, e.g., the intended purchase price, could lead to financial disadvantages, e.g., due to insider trading. To support the described scenario, the user can exploit statement-oblivious threshold witness encryption in the following way. The user encrypts its transaction with the desired share price as statement and a signature of a trusted price oracle service as the required witnesses. Trusted price oracles are already available in the DeFi ecosystem and heavily used for building various financial products. The ciphertext is sent to the user’s broker who repeatedly requests the signed current share price from the oracle service, attempts decryption, and, if this gives a valid transaction, executes the trade. For decrypting, the broker sends the ciphertext together with the current share price to the decryption committee. As the statement is hidden, no one, not even the oracle service, learns the desired share price until the transaction is successfully decrypted. Due to the required signature of the oracle service, the broker cannot send incorrect share prices to the decryption committee. We provide more details about use-cases of our new security feature in Section IX.

While [8] and [9] tackled the first limitation and present more efficient constructions that are effectively the same as witness encryption, both schemes still suffer from the fact that the statement is public. In this work, we address the privacy feature mentioned above. To this end, we introduce a novel notion that we call *statement-oblivious threshold witness encryption (SO-TWE)* and show how to instantiate it.

A. Contribution

We start by giving a summary of our contribution and defer a high-level overview of our constructions as well as a discussion of the technical challenges to the technical overview.

1) *Primitive definition*: We introduce the notion of *statement-oblivious threshold witness encryption (SO-TWE)*. This primitive provides effectively the same functionality as witness encryption while requiring a committee with a fixed number of corrupted parties as typically done in threshold cryptography. As we envision the committee to perform decryption on request, we define a *security notion against*

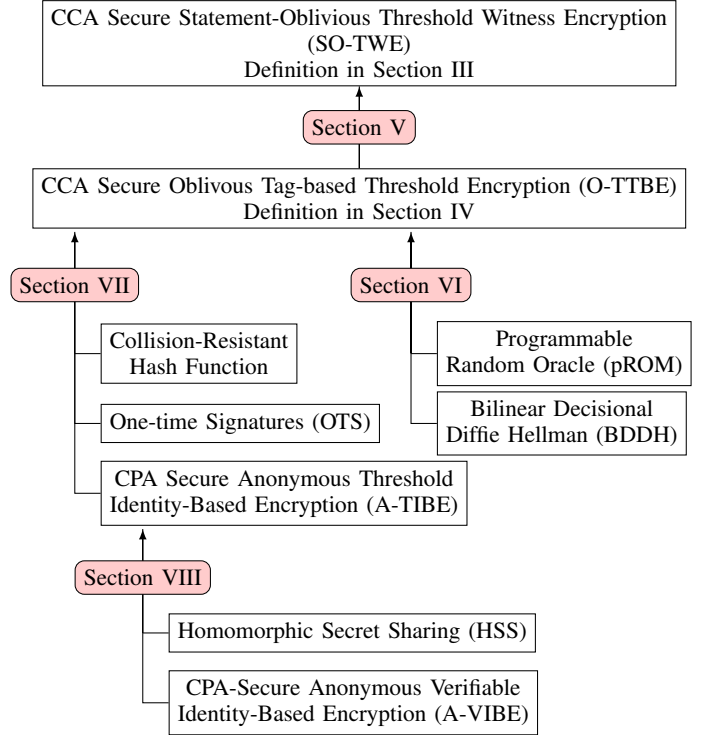


Fig. 1. The Landscape of Our Contributions.

chosen-ciphertext attacks (CCA) which is at least as strong as the notion of extractability for threshold witness encryption. In addition, the *statement-obliviousness* property guarantees that the statement used to generate a ciphertext is hidden. We provide a formal security game combining the CCA security with our new statement-obliviousness property.

We do not follow up on the existing notions of extractable Witness Encryption on Blockchains, proposed by [8], or Blockchain Witness Encryption, proposed by [10], as both notions are tied to the blockchain setting. We take a more general approach by using the committee to achieve witness encryption without defining the origin of the committee. In contrast to earlier works, however, our notion considers only static corruptions.

2) *Instantiating SO-TWE*: We show how to instantiate SO-TWE via a series of transformations, as depicted in Figure 1. For all constructions and transformations, we provide formal security proofs. As a first step, we introduce the notion of *oblivious threshold tag-based encryption (O-TTBE)* as an extension of standard threshold tag-based encryption as presented in [15]. Similar to statement-obliviousness, obliviousness in this context ensures that the tag used for encryption is hidden. Then, we present a general transformation from CCA secure O-TTBE to CCA secure SO-TWE.

As a second step, we show two ways to construct CCA secure O-TTBE schemes. First, we generically build a O-TTBE scheme from collision-resistant hash functions, one-time signatures and CPA secure anonymous threshold identity-based encryption (A-TIBE). To the best of our knowledge, there are

constructions for anonymous identity-based encryption [16], [17] and threshold identity-based encryption [18], but there is no construction of an A-TIBE scheme. The techniques used for anonymous IBE do not allow for a straightforward thresholdization via secret sharing while maintaining a high threshold and non-interactive decryption at the same time. As a feasibility result, we show how to instantiate A-TIBE from non-threshold anonymous identity-based encryption, a signature scheme and homomorphic secret sharing (HSS). This transformation follows [10] which constructs non-anonymous threshold identity-based encryption from HSS. While proving the security of our construction, we discovered a gap in the analysis of [10]. In particular, the construction in [10] allows corrupted parties to trick honest parties into accepting invalid identity keys, and hence, does not provide *key generation consistency*. We propose a solution to fix this gap. While the A-TIBE-based construction constitutes a feasibility result, we emphasize that any progress in constructing these building blocks, e.g., in terms of efficiency, immediately yields more efficient constructions of SO-TWE.

As a second way, we present a concretely efficient instantiation of O-TTBE in the random oracle model. Our construction extends Hash-ElGamal with a bilinear mapping and efficient non-interactive zero knowledge arguments. The resulting scheme is concretely efficient in terms of ciphertext size and bilinear mapping evaluations. The construction also yields the first efficient threshold witness encryption scheme that additionally achieves statement obliviousness. This is because our generic transformation from O-TTBE to SO-TWE only adds simple hash function evaluations and a check of the witness relation. We formally prove the security of this construction via a reduction to the *Decision Bilinear Diffie-Hellman assumption*.

B. Technical Overview

In this section, we outline the main techniques used to construct SO-TWE and discuss the major challenges.

Emulation of the witness encryption functionality. We consider the setup of a SO-TWE scheme to be executed by a trusted dealer or via a distributed key generation protocol. During the setup, the public key and the verification key are published while the secret key shares are distributed to the committee members. It is assumed that an adversary can statically corrupt a subset of the committee members. We allow the adversary to corrupt all but one committee member. Upon corruption the adversary takes full control over the committee members, and hence, learns their secret key shares. Users can encrypt messages non-interactively based on the public key and a self-chosen statement. Decryption is performed in an interactive way via a request-response protocol. To this end, a user sends the ciphertext, a statement candidate and a witness to the committee. All committee members compute and send their decryption shares to the user, who attempts to combine the shares to the actual message. This will only be successful if it receives sufficiently many valid decryption shares and the witness relation has been verified successfully. Further, the

statement-obliviousness property provides that the combined shares will only yield the original plaintext if the statement candidate used for the decryption is the same as the one used for encryption. While emulating the functionality of witness encryption using a committee-based approach seems to be easy at first glance, achieving statement obliviousness in combination with CCA security is highly non-trivial, as discussed next.

Achieving obliviousness in the CCA-setting. Due to the committee setting in which decryption is executed on request, we require security against chosen-ciphertext attacks (CCA). The major challenge is to simultaneously guarantee CCA security and achieve our new notion of statement obliviousness. A common technique to achieve CCA security in the threshold setting is to incorporate ciphertext validation before decryption [18]–[22]. The validation ensures that each decryption request issued by the adversary in the security game is either declined or yields exactly the original plaintext created by some user. This feature is required by the security proofs of known CCA secure threshold constructions, e.g., to prevent the adversary from exploiting homomorphisms in the group structure to decrypt valid ciphertexts that contain related messages. The difficulty in our setting is that the decryption committee may not know the statement used for encryption. In fact, the information if the correct statement has been used for decryption must not be leaked before decryption is completed. Any such leakage would allow corrupted servers to break the obliviousness property. It follows that we have to allow for multiple decryptions, with different statements, of the same ciphertext, and hence, cannot follow the standard approach of previous work. The described scenario makes it highly challenging to achieve obliviousness in combination with CCA security in the threshold setting. In particular, the challenge is to render decryptions useless for statements different than the one used for encryption despite applying the correct secret key shares when generating the decryption shares. Prior CCA-secure encryption schemes apply the secret key (shares) during decryption only after ensuring that the resulting (combined) decryption yields exactly the original message. Hence, we cannot use existing approaches to solve the described challenge.

SO-TWE from oblivious threshold tag-based encryption (cf. Section V). As a first step towards SO-TWE, we present a transformation from a primitive called *oblivious threshold tag-based encryption (O-TTBE)*. To this end, we first extend the standard notion of threshold tag-based encryption presented by Arita and Tsurudome [15] with an obliviousness property. Similar to statement-obliviousness, obliviousness for a tag-based encryption scheme requires that two ciphertexts created with different tags cannot be distinguished.

Our first transformation takes a CCA secure O-TTBE scheme in order to construct SO-TWE. The high-level idea is to use the hash of the statement as a tag for the O-TTBE scheme. For decryption, a user needs to provide a statement candidate together with a corresponding witness. The decryption servers first check if the witness is valid and

then use the hash of the provided statement candidate as the tag in the decryption of the O-TTBE scheme. The statement-obliviousness property is directly obtained from the obliviousness property of the O-TTBE scheme but constructing CCA secure O-TTBE still faces the challenges explained above. As depicted in Figure 1, we follow two different paths to overcome these challenges and to construct CCA secure O-TTBE as described below.

O-TTBE from programmable random oracles and bilinear maps (cf. Section VI). In general, independently of the obliviousness setting, the major difficulty when proving CCA security is to answer decryption queries without knowledge of the secret key. When instantiating O-TTBE from black-box primitives, this task is realized by using oracles of the underlying primitive in the reduction. For example, in our transformation from O-TTBE to SO-TWE the reduction to O-TTBE uses the decryption oracle of the O-TTBE security game to answer decryption queries of the SO-TWE adversary. When combining CCA security with an obliviousness property, we additionally face the discussed challenge to answer different decryption queries for the same ciphertext. Here, a random looking value needs to be returned except for the decryption query that contains the tag used for encryption. For a concrete O-TTBE scheme, we need to address both challenges in parallel. Due to the strict ciphertext validation used in existing CCA secure encryptions schemes (e.g., [18]–[22]) extending these schemes to support tag obliviousness cannot be done in a straightforward way.

We propose a new construction starting from CPA secure Hash-ElGamal, which is a variant of classical ElGamal [23]. In Hash-ElGamal, the encryption algorithm given a message m samples a random exponent a and outputs two elements $A = g^a$ and $M = m \oplus H(X^a)$ for a group generator g , a random oracle H , and a public key $X = g^x$. In the threshold setting, the secret key x is secret shared among the decryption servers. The decryption shares of the servers are calculated as $d_i := A^{x_i}$, where x_i is the share of the i -th server. We apply an extension to this scheme that allows us to solve both aforementioned challenges at once. We do so by applying a random offset T to A in both, encryption and decryption. This offset is unique for each ciphertext-tag pair to obtain random values from decryption for tags different to the one used for encryption. When applying the offset via multiplication or exponentiation, e.g., $M = m \oplus H(X^{a \cdot T})$, an adversary can easily perform an homomorphic attack, i.e., $A^{x_i \cdot T} = (A^{x_i})^T$. In order to prevent this, we apply the offset using a bilinear mapping e , i.e., $M = m \oplus H(e(T, X^a))$.

Further, we ensure that a ciphertext component A cannot be reused in different ciphertexts except by the party that generated A , and hence, knows the plaintext anyway. We do so, by adding a non-interactive zero-knowledge argument of knowledge of a to the ciphertext. The second ciphertext component, M , is used for computing the challenge value of the non-interactive zero-knowledge argument, in order to link this component to the zero-knowledge argument. In classical ElGamal-based schemes, adding a zero-knowledge argument

of knowledge of a to the ciphertext is not sufficient to achieve CCA security, as demonstrated in detail by [19]. Instead, it is necessary to provide an additional trapdoor to solve the general challenge of CCA security, to answer decryption queries. Interestingly, in our construction, the tag-dependent offset does not only give us tag obliviousness but also provides us with such a trapdoor for free. In particular, in the reduction, we can simulate the random oracle used to compute the offset such that we learn the discrete logarithm of all offsets sampled by the random oracle. This allows us to compute $e(T, X^a)$ via $e(A, X)^{\log_g(T)}$. We elaborate further on the concrete challenges and the intuition of our construction in Section VI before presenting the formal specification.

Despite being the first instantiation of O-TTBE, our construction yields a concretely efficient scheme. The ciphertexts consist of a bitstring with length equal to the message length, a group element of the bilinear mapping's base group and two exponents (in \mathbb{Z}_q , where q is the bilinear group's order). Decryption shares consist of one group element in the mapping's target group and two exponents. Encryption requires a single evaluation and decryption three evaluations of the bilinear map.

O-TTBE from anonymous threshold identity-based encryption (cf. Section VII). While the construction described in the previous paragraph yields an efficient scheme, we also present a generic solution. Boneh et al. [18] show how to achieve CCA security from one-time signatures and CPA secure identity-based encryption. Following this approach, we achieve CCA security in the threshold setting by combining one-time signatures with CPA secure *anonymous threshold identity-based encryption* (A-TIBE). The anonymity property of the TIBE is utilized to achieve obliviousness of the TTBE scheme. The high-level idea is to encode the tag into the identity of the IBE ciphertext. Since the anonymity property guarantees that no information about the identity can be obtained from the ciphertext, the tag stays hidden as well. Only the decryption with the correct tag, i.e., with the identity key corresponding to the tag, reveals information about the plaintext.

Constructing A-TIBE (cf. Section VIII). As a final step, we explore two directions to obtain anonymous threshold identity-based encryption (A-TIBE). First, we present a black-box construction based on homomorphic secret sharing (HSS). The same approach was used by Campanelli et al. [10] in order to construct threshold IBE without anonymity. When exploring this direction, we discovered a gap in the security analysis of [10]. The construction in [10] does not provide *key generation consistency*, a security property that enables parties to validate correctness of received identity keys. Without that property, maliciously corrupted committee members can provide arbitrary identity key shares. This may result in an incorrect identity key such that the decryption of some ciphertext yields a different plaintext than the originally encrypted message. As such an attack is not possible in the non-threshold setting, standard IBE does not provide means to validate identity keys. It follows that the straightforward thresholdization of IBE using HSS is not sufficient to provide a secure threshold IBE

scheme.

To overcome this problem, we propose a new IBE primitive with an additional verifiability property. Verifiable IBE contains a check if an identity key is computed correctly which may be of independent interest in other settings where malicious security is required. Such a scheme can be built from a standard IBE scheme together with an existentially unforgeable signature scheme. Eventually, we construct anonymous threshold IBE by executing the key generation algorithm of the verifiable IBE scheme within HSS. We provide a formal proof showing security of the construction, including the discussed identity key generation consistency property. We note that in this black-box construction, we need to consider general-purpose HSS like [10].

Finally, we explore the transformation of the concrete anonymous non-threshold IBE scheme of Boyen and Waters [16]. The challenge in this transformation is that the identity key generation requires multiplication of secret values and freshly chosen randomness that needs to remain private. A direct secret sharing of these values pose some challenges which we discuss in the full version of this paper [24]. While general-purpose secure multi-party computation can solve this task, we aim for a threshold IBE scheme that requires no interaction during identity key generation. We point out and discuss two ways how the aforementioned issues can be tackled and leave formal specifications and security analyses of these approaches to future work.

II. PRELIMINARIES

Here, we present the most important primitives. Throughout this work, we denote the security parameter by $\kappa \in \mathbb{N}$. We denote the set $\{1, \dots, k\}$ as $[k]$. For a negligible function $\text{negl} : \mathbb{N} \rightarrow \mathbb{R}$, it holds that for every $c \in \mathbb{N}$ there exists a $n_0 \in \mathbb{N}$ such that for all $n > n_0$: $|\text{negl}(n)| < \frac{1}{n^c}$. For the sake of expressiveness, we often denote a negligible function by negl . We use the abbreviation PPT to denote a probabilistic polynomial-time algorithm.

A. Bilinear Maps

We briefly recall the basics of bilinear maps following [18], [25]. Let BGen be a randomized algorithm that on input a security parameter κ outputs a prime q , such that $\log_2(q) = O(\kappa)$, two cyclic groups of prime order q and a pairing $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$.

We call e a bilinear map if the following properties hold:

- **Bilinearity:** For all $u, v \in \mathbb{G}$ and $a, b \in \mathbb{Z}_q$, we have $e(u^a, v^b) = e(u, v)^{ab}$.
- **Non-degeneracy:** For generator g of \mathbb{G} it holds that $e(g, g) \neq 1$. Since \mathbb{G}_T is of prime order q , this implies that $e(g, g)$ is a generator of \mathbb{G}_T .
- **Efficiency:** e can be computed efficiently in polynomial time in κ .

A bilinear map satisfying the above properties is sometimes called *admissible* bilinear map. We are only interested in admissible bilinear maps and implicitly mean this type of bilinear maps when writing bilinear maps in short. We call

BGen a *Bilinear Group Generator* if the algorithm can be computed efficiently in polynomial time in κ and each pairing e generated by BGen is a bilinear map.

While in the above setting the decisional Diffie-Hellman assumption (DDH) does not hold in group \mathbb{G} , there is an extension to the setting with bilinear maps.

Definition 1 (DBDH). *The Decision Bilinear Diffie-Hellman assumption (DBDH) states that for every Bilinear Group Generator BGen and algorithm \mathcal{D} running in time polynomial in security parameter κ it holds that*

$$\left| \Pr[\mathcal{D}(\bar{\mathbb{G}}, g, h, g^a, g^b, e(h, g)^{ab})] - \Pr[\mathcal{D}(\bar{\mathbb{G}}, g, h, g^a, g^b, R)] \right| \leq \text{negl}(\kappa)$$

where $\bar{\mathbb{G}} = (q, \mathbb{G}, \mathbb{G}_T, e) \leftarrow_R \text{BGen}(\kappa)$, $g, h \in_R \mathbb{G}$, $R \in_R \mathbb{G}_T$, and $a, b, c \in_R \mathbb{Z}_q$. The randomness is taken over the random choices of BGen, the group elements g, h, R , the values a, b, c , and the random bits of \mathcal{D} .

B. Hash Functions and Digital Signatures

A hash function H is a function that takes as input a string $x \in \{0, 1\}^*$ and returns a fixed-length output string $H(x) \in \{0, 1\}^{\ell(\kappa)}$ for some polynomial $\ell(\kappa)$. A signature scheme $\text{SIG} = (\text{KeyGen}, \text{Sign}, \text{Verify})$ over message space \mathbb{M} consists of three probabilistic polynomial-time algorithms. The key generation algorithm KeyGen produces a key pair $(\text{SigK}, \text{VerK})$ on security parameter 1^κ . The signing algorithm Sign takes a signing key SigK and a message $m \in \mathbb{M}$ and produces a signature σ . A signature σ on message m can be verified with respect to the verification key VerK using the verification algorithm Verify. As standard, we require the hash function to satisfy *collision resistance* and the digital signature scheme to provide *consistency* and *existential unforgeability against chosen-message attacks*. Formal definitions of these properties are provided in Appendix B-A and B-B.

C. Anonymous Threshold Identity-Based Encryption

We derive the notion of *Anonymous Threshold Identity-Based Encryption* from [17] as follows:

Definition 2 (TIBE). *An anonymous threshold identity-based encryption scheme (TIBE) TIBE is associated with the following probabilistic polynomial-time algorithms:*

- 1) $\text{Setup}(1^\kappa, s, n)$ takes as input a security parameter 1^κ , the number of decryption servers n and the security threshold s , with $1 \leq s \leq n$. It generates system parameters pk , a verification key vk , and n master secret key shares $\{\text{sk}_i\}_{i \in [n]}$. The i -th decryption server gets master secret key share sk_i .
- 2) $\text{ShareKeyGen}(\text{pk}, i, \text{sk}_i, \text{id})$ takes as input the public parameter pk , the decryption server index i , the corresponding secret key sk_i and an identity $\text{id} \in \{0, 1\}^*$. It generates an identity key share (i, ik_i) .
- 3) $\text{ShareVf}(\text{pk}, \text{vk}, \text{id}, i, \text{ik}_i)$ takes as input the public parameter pk , the verification key vk , an identity id , a decryption server index i and an identity key share ik_i . It outputs true or false.

- 4) $\text{Combine}(\text{pk}, \text{vk}, \text{id}, \{(i, \text{ik}_i)\}_{i \in \mathcal{S}})$ takes as input the public parameter pk , the verification key vk , an identity id and indexed identity key shares ik_i and returns an identity key ik or \perp .
- 5) $\text{Encrypt}(\text{pk}, \text{id}, m)$ takes as input the public parameter pk , an identity id and a message m and outputs a ciphertext c .
- 6) $\text{Decrypt}(\text{pk}, \text{id}, \text{ik}, c)$ takes as input the public parameter pk , an identity id , an identity key ik and a ciphertext c and outputs a message m .

We require for all $\kappa, n, s \in \mathbb{N}$, where $1 \leq s \leq n$, and any $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]}) \leftarrow \text{Setup}(1^\kappa, s, n)$ the following properties:

- **Share consistency:** For any identity $\text{id} \in \{0, 1\}^*$ and any $i \in [n]$, if $(i, \text{ik}_i) \leftarrow \text{ShareKeyGen}(\text{pk}, i, \text{sk}_i, \text{id})$, then $\text{ShareVf}(\text{pk}, \text{vk}, \text{id}, i, \text{ik}_i) = \text{true}$.
- **Decryption correctness:** For any identity $\text{id} \in \{0, 1\}^*$, if \mathcal{S} is a subset of $[n]$ of size s , $\mathcal{IK} := \{(i, \text{ik}_i) \mid i \in \mathcal{S} \wedge (i, \text{ik}_i) \leftarrow \text{ShareKeyGen}(\text{pk}, i, \text{sk}_i, \text{id})\}_{i \in \mathcal{S}}$, and $\text{ik} \leftarrow \text{Combine}(\text{pk}, \text{vk}, \text{id}, \mathcal{IK})$, then we require that for any m in the message space, $m = \text{Decrypt}(\text{pk}, \text{ik}, \text{Encrypt}(\text{pk}, \text{id}, m))$.

Security. We define security via three properties: *key generation consistency*, *security against chosen-identity attacks* and *anonymity*. Informally, the first one states that an adversary cannot generate a ciphertext and two sets of valid identity key shares for the same identity such that the shares combine to different keys and the ciphertext is decrypted to two different plaintexts. The last ones state that an adversary cannot distinguish between two encryptions and two identities used for encryption. We formally define the security game and ANON-IND-ID-CPA security in Appendix B-E.

III. STATEMENT-OBLIVIOUS THRESHOLD WITNESS ENCRYPTION

In the setting of *threshold witness encryption (TWE)*, we distinguish between users and decryption servers. Users either aim to encrypt some plaintext under a statement x in some NP language \mathcal{L} or aim to decrypt some ciphertext knowing a witness corresponding to the statement $x \in \mathcal{L}$. Decryption servers possess private information and assist users while decrypting a ciphertext. The decryption servers constitute a committee with a fixed number of corrupted parties. The committee may be static or adaptive depending on the concrete instantiation. For instance, a line of work [10], [13], [14] proposed mechanisms to select committees without revealing the identity of the members until they speak to protect against adaptive adversaries. The constructions are based on techniques incorporated in many popular blockchain. We emphasize that our definition and construction abstracts from the concrete instantiation of the committee. We only assume that a committee consists of n decryption servers and only $s - 1$ of them are corrupted. Moreover, we assume the setup procedure of a TWE construction to be executed by a trusted dealer. This approach is standard in threshold cryptography

and a trusted dealer could be realized by a tailored multi-party computation protocol. The dealer distributes secret information to the decryption servers and publishes public information to all parties.

In contrast to the definition of extractable witness encryption on blockchain (eWEB) by [8], we abstract away the realization of the committee while their definition explicitly considers a dynamic committee and a hand-off procedure to move from one committee to another. Since the change of the committee members is inherent to their definition, they also consider adaptive corruption in their security game. Moreover, their definition specifically considers a model where plaintexts are shared to the committee members which reveal these information only if a witness is presented. In contrast, our definition follows the approach presented by [9] where only a single secret key is shared between the committee members. In contrast to the definition of blockchain witness encryption (BWE) by Campanelli et al. [10] we do not explicitly define our TWE based notion for blockchains. Here again, we abstract away the concrete realization of the committee.

Formally, we define our new primitive as follows.

Definition 3 (TWE). A threshold witness encryption scheme (TWE) TWE for an NP language \mathcal{L} with associated relation R consists of the following five PPT algorithms:

- 1) $\text{Setup}(1^\kappa, s, n)$ takes as input the security parameter 1^κ , a threshold s , and the number of decryption servers n , where $1 \leq s \leq n$. It outputs a triple $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]})$, where pk is a public key, vk is a verification key, and sk_i is the secret key share for the decryption server with index i .
- 2) $\text{Encrypt}(\text{pk}, x, m)$ takes as input the public key pk , a statement x , and a message m . It outputs a ciphertext c .
- 3) $\text{ShareDec}(\text{pk}, c, x, w, (i, \text{sk}_i))$ takes as input a public key pk , a ciphertext c , a statement x , a witness w , and the index i together with the secret key share sk_i of the i -th decryption server. It outputs a decryption share d_i or a failure symbol \perp together with the index i .
- 4) $\text{ShareVf}(\text{pk}, \text{vk}, c, x, (i, d_i))$ takes as input a public key pk , a verification key vk , a ciphertext c , a statement x , and an indexed decryption share (i, d_i) . It outputs false if the decryption share is invalid and true if it is valid with respect to pk , vk , c , and x .
- 5) $\text{Combine}(\text{pk}, \text{vk}, c, x, \{(i, d_i)\}_{i \in \mathcal{S}})$ takes as input a public key pk , a verification key vk , a ciphertext c , a statement x , and a set of decryption shares $\{(i, d_i)\}_{i \in \mathcal{S}}$. It outputs message m or \perp .

We require for every security parameter $\kappa \in \mathbb{N}$, every NP-language \mathcal{L} with associated relation R , every $n, s \in \mathbb{N}$ where $1 \leq s \leq n$, every output $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]})$ of $\text{Setup}(1^\kappa, s, n)$, every $x \in \mathcal{L}$ and w such that $(x, w) \in R$, for every message m , and every ciphertext $c \leftarrow \text{Encrypt}(\text{pk}, x, m)$:

- **Decryption share validity:** If $(i, d_i) \leftarrow \text{ShareDec}(\text{pk}, c, x, w, (i, \text{sk}_i))$, then $\text{ShareVf}(\text{pk}, \text{vk}, c, x, (i, d_i)) = 1$.

- **Correctness:** For any $\mathcal{S} \subseteq [n]$ of size s , if $\{(i, d_i)\}_{i \in \mathcal{S}}$ is a set of distinct decryption shares with $(i, d_i) \leftarrow \text{ShareDec}(\text{pk}, c, x, w, (i, \text{sk}_i))$ for each $i \in \mathcal{S}$, then $\text{Combine}(\text{pk}, \text{vk}, c, x, \{(i, d_i)\}_{i \in \mathcal{S}}) = m$.

Security. We define security via three properties: *indistinguishability under chosen-ciphertext attacks* (IND-CCA), *statement obliviousness* (SO) and *decryption consistency under chosen-ciphertext attacks* (DC-CCA). Intuitively, IND-CCA and SO state that ciphertexts created using two different messages and two different statements cannot be distinguished. We combine these property formally in the security game $\text{Exp}^{\text{SO-CCA}}$. The DC-CCA property states that an adversary cannot produce two sets of valid decryption shares that are combined to two different messages unequal \perp . Formally, we define the security game $\text{Exp}^{\text{SO-DC}}$.

```

Experiment  $\text{Exp}_{\text{TWE}, \mathcal{A}}^{\text{SO-CCA}}(1^\kappa)$ 
-----
 $\mathcal{M} \leftarrow \mathcal{A}_0(1^\kappa)$  with  $|\mathcal{M}| < s$ 
 $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]}) \leftarrow \text{Setup}(1^\kappa, s, n)$ 
 $\alpha, \beta \in_R \{0, 1\}$ 
 $(x_0, x_1, m_0, m_1) \leftarrow \mathcal{A}_1^{\mathcal{O}(\cdot, \cdot, \cdot)}(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in \mathcal{M}})$ 
 $c^* \leftarrow \text{Encrypt}(\text{pk}, x_\alpha, m_\beta)$ 
 $(\alpha', \beta') \leftarrow \mathcal{A}_2^{\mathcal{O}(\cdot, \cdot, \cdot)}(c^*)$ 
return  $(\alpha, \beta) = (\alpha', \beta')$ 

```

In the given security game, the adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$ corrupts the decryption servers in \mathcal{M} . \mathcal{A}_1 and \mathcal{A}_2 can use the oracle $\mathcal{O}(\cdot, \cdot, \cdot)$ to make decryption queries. To do so, the adversary sends (i, c, x, w) to \mathcal{O} which returns $(i, d_i) \leftarrow \text{ShareDec}(\text{pk}, c, x, w, (i, \text{sk}_i))$. Only for \mathcal{A}_2 , the oracle first checks if $c = c^*$, $x \in \{x_0, x_1\}$ and $(x, w) \in R$. If this holds, the oracle returns (i, \perp) and otherwise it returns a correct decryption share.

```

Experiment  $\text{Exp}_{\text{TWE}, \mathcal{A}}^{\text{SO-DC}}(1^\kappa)$ 
-----
 $\mathcal{M} \leftarrow \mathcal{A}_0(1^\kappa)$  with  $|\mathcal{M}| < s$ 
 $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]}) \leftarrow \text{Setup}(1^\kappa, s, n)$ 
 $(x, c, \{(i, d_i)\}_{i \in \mathcal{S}}, \{(i, d'_i)\}_{i \in \mathcal{S}'}) \leftarrow \mathcal{A}_1^{\mathcal{O}(\cdot, \cdot, \cdot)}(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in \mathcal{M}})$ 
 $m \leftarrow \text{Combine}(\text{pk}, \text{vk}, c, x, \{(i, d_i)\}_{i \in \mathcal{S}})$ 
 $m' \leftarrow \text{Combine}(\text{pk}, \text{vk}, c, x, \{(i, d'_i)\}_{i \in \mathcal{S}'})$ 
  where  $\mathcal{S}, \mathcal{S}' \subseteq [n] \wedge |\mathcal{S}| = s = |\mathcal{S}'|$ 
if  $\forall i \in \mathcal{S} : \text{ShareVf}(\text{pk}, \text{vk}, c, x, (i, d_i)) = \text{true}$ 
   $\wedge \forall i \in \mathcal{S}' : \text{ShareVf}(\text{pk}, \text{vk}, c, x, (i, d'_i)) = \text{true}$ 
   $\wedge \perp \neq m \neq m' \neq \perp$ 
  return 1
else
  return 0

```

Here, the adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ corrupts the decryption servers in \mathcal{M} and \mathcal{A}_1 can use the decryption oracle $\mathcal{O}(i, c, x, w)$ that returns $(i, d_i) \leftarrow \text{ShareDec}(\text{pk}, c, x, w, (i, \text{sk}_i))$.

Definition 4 (SO-IND-CCA Security of TWE). A *threshold witness encryption scheme* TWE is *statement-oblivious*

and message-indistinguishable under chosen-ciphertext attacks (SO-IND-CCA) secure if for all PPT adversaries $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$, there exist negligible functions negl_0 and negl_1 such that

$$\left| \Pr[\text{Exp}_{\text{TWE}, \mathcal{A}}^{\text{SO-CCA}}(1^\kappa) = 1] - \frac{1}{4} \right| \leq \text{negl}_0(\kappa) \quad \wedge$$

$$\Pr[\text{Exp}_{\text{TWE}, \mathcal{A}}^{\text{SO-DC}}(1^\kappa) = 1] \leq \text{negl}_1(\kappa).$$

a) *Remark 1:* The standard notion of witness encryption (cf. [1]) defines security without access to a decryption oracle. This is due to the fact that decryption in the standard notion can be attempted by any party locally using knowledge of the witness. In the threshold setting, decryption is performed via an interaction with a decryption committee that performs decryption in a distributed way using a secret shared trapdoor. Hence, we have to give the adversary access to a decryption oracle.

b) *Remark 2:* We note that in the context of TWE SO-IND-CCA security implies *extractability*, an additional security requirement often required from witness encryption. We provide further details to the notion of extractability for TWE and a reduction from extractability to SO-IND-CCA security in Appendix A.

IV. OBLIVIOUS THRESHOLD TAG-BASED ENCRYPTION

In this section, we present the notion of *oblivious threshold tag-based encryption* (O-TTBE) which constitutes an extension of standard threshold tag-based encryption as presented in [15]. Intuitively, a threshold tag-based encryption scheme is *oblivious* if a ciphertext hides the tag it was created with. We first state the definition of threshold tag-based encryption and present the obliviousness property as part of the security guarantees afterwards.

Definition 5 (TTBE). A threshold tag-based encryption scheme (TTBE) TTBE consists of the following five PPT algorithms:

- 1) $\text{Setup}(1^\kappa, s, n)$ takes as input the security parameter 1^κ , a threshold s , and the number of decryption servers n where $1 \leq s \leq n$. It outputs a triple $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]})$, where pk is a public key, vk is a verification key, and sk_i is the secret key share for the i -th decryption server.
- 2) $\text{Encrypt}(\text{pk}, t, m)$ takes as input a public key pk , a tag t , and a message m , and it outputs a ciphertext c .
- 3) $\text{ShareDec}(\text{pk}, c, t, (i, \text{sk}_i))$ takes as input a public key pk , a ciphertext c , a tag t , and the index i together with the secret key share sk_i of the decryption server with index i . It outputs a decryption share d_i or a failure symbol \perp together with the index i .
- 4) $\text{ShareVf}(\text{pk}, \text{vk}, c, t, (i, d_i))$ takes as input a public key pk , a verification key vk , a tag t , and an indexed decryption share (i, d_i) . It outputs false if the decryption share is invalid and true if it is valid with respect to pk , vk , c and t .
- 5) $\text{Combine}(\text{pk}, \text{vk}, c, t, \{(i, d_i)\}_{i \in \mathcal{S}})$ takes as input a public key pk , a verification key vk , a ciphertext c , a tag t , and a

set of decryption shares $\{(i, d_i)\}_{i \in \mathcal{S}}$. It outputs message m or \perp .

We require for every security parameter $\kappa \in \mathbb{N}$, every committee parameters $n, s \in \mathbb{N}$ where $1 \leq s \leq n$, every $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]})$ generated by $\text{Setup}(1^\kappa, s, n)$, every message m , every tag t and every $c \leftarrow \text{Encrypt}(\text{pk}, t, m)$:

- **Decryption share validity:** If $(d_i, i) \leftarrow \text{ShareDec}(\text{pk}, c, t, (i, \text{sk}_i))$, then $\text{ShareVf}(\text{pk}, \text{vk}, c, t, (i, d_i)) = 1$.
- **Correctness:** If $\{(i, d_i)\}_{i \in \mathcal{S}}$ is a set of s distinct decryption shares with $(i, d_i) \leftarrow \text{ShareDec}(\text{pk}, c, t, (i, \text{sk}_i))$ for each $i \in \mathcal{S}$, then $\text{Combine}(\text{pk}, \text{vk}, c, t, \{(i, d_i)\}_{i \in \mathcal{S}}) = m$.

Security. Security of a TTBE scheme is defined via two properties: *oblivious indistinguishable messages under chosen-ciphertext attacks* (IND-CCA) and *decryption consistency under chosen-ciphertext attacks* (DC-CCA). The intuition for these properties is analog to the ones of threshold witness encryption. The IND-CCA property states that ciphertexts created using two different messages and two different tags cannot be distinguished. The DC-CCA property states that an adversary cannot produce two sets of valid decryption shares that are combined to two different messages unequal \perp . To formalize these properties, we design the following security games:

Experiment $\text{Exp}_{\text{TTBE}, \mathcal{A}}^{\text{O-CCA}}(\kappa)$

$\mathcal{M} \leftarrow \mathcal{A}_0(1^\kappa)$ with $|\mathcal{M}| < s$
 $\alpha, \beta \in_R \{0, 1\}$
 $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]}) \leftarrow \text{Setup}(1^\kappa, s, n)$
 $(t_0, t_1, m_0, m_1) \leftarrow \mathcal{A}_1^{\text{O}(\cdot, \cdot, \cdot)}(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in \mathcal{M}})$
 $c^* \leftarrow \text{Encrypt}(\text{pk}, t_\alpha, m_\beta)$
 $(\alpha', \beta') \leftarrow \mathcal{A}_2^{\text{O}(\cdot, \cdot, \cdot)}(c^*)$
return $(\alpha, \beta) = (\alpha', \beta')$

The decryption oracle $\mathcal{O}(\cdot, \cdot, \cdot)$ takes as parameter an index i , a ciphertext c and a tag t , and computes $(i, d_i) \leftarrow \text{ShareDec}(\text{pk}, c, t, (i, \text{sk}_i))$. If $(c, t) \in \{(c^*, t_0), (c^*, t_1)\}$ it returns (i, \perp) , otherwise it returns (i, d_i) .

Experiment $\text{Exp}_{\text{TTBE}, \mathcal{A}}^{\text{O-DC}}(\kappa)$

$\mathcal{M} \leftarrow \mathcal{A}_0(1^\kappa)$ with $|\mathcal{M}| < s$
 $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]}) \leftarrow \text{Setup}(1^\kappa, s, n)$
 $(t, c, \{(i, d_i)\}_{i \in \mathcal{S}}, \{(i, d'_i)\}_{i \in \mathcal{S}'}) \leftarrow \mathcal{A}_1^{\text{O}(\cdot, \cdot, \cdot)}(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in \mathcal{M}})$
 where $\mathcal{S}, \mathcal{S}' \subseteq [n] \wedge |\mathcal{S}| = s = |\mathcal{S}'|$
 $m \leftarrow \text{Combine}(\text{pk}, \text{vk}, c, t, \{(i, d_i)\}_{i \in \mathcal{S}})$
 $m' \leftarrow \text{Combine}(\text{pk}, \text{vk}, c, t, \{(i, d'_i)\}_{i \in \mathcal{S}'})$
if $\forall i \in \mathcal{S} : \text{ShareVf}(\text{pk}, \text{vk}, c, t, (i, d_i)) = \text{true}$
 $\wedge \forall i \in \mathcal{S}' : \text{ShareVf}(\text{pk}, \text{vk}, c, t, (i, d'_i)) = \text{true}$
 $\wedge \perp \neq m \neq m' \neq \perp$
return 1
else
return 0

The decryption oracle $\mathcal{O}(\cdot, \cdot, \cdot)$ takes as parameter an index i , a ciphertext c and a tag t , and returns $\text{ShareDec}(\text{pk}, c, t, (i, \text{sk}_i))$.

Definition 6. A TTBE scheme TTBE is *OB-IND-CCA secure* if for every PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$, there exists negligible functions negl_0 and negl_1 such that

$$\left| \Pr[\text{Exp}_{\text{TTBE}, \mathcal{A}}^{\text{O-CCA}}(\kappa) = 1] - \frac{1}{4} \right| \leq \text{negl}_0(\kappa) \wedge \Pr[\text{Exp}_{\text{OTTBE}, \mathcal{A}}^{\text{O-DC}}(\kappa) = 1] \leq \text{negl}_1(\kappa).$$

We use the notation of *oblivious TTBE* in short for referring to an OB-IND-CCA secure TTBE.

V. CONSTRUCTING STATEMENT-OBLIVIOUS TWE

In this section, we present a construction for statement-oblivious threshold witness encryption (SO-TWE) from oblivious threshold tag-based encryption (O-TTBE).

Construction 1: SO-TWE_{OTTBE}

Public parameters:

The scheme is defined for a language \mathcal{L} with relation R . The number of committee members is denoted by n and the threshold parameter is s . We make use of a oblivious tag-based encryption scheme OTTBE and a collision-resistant hash function $H : \mathbb{X} \rightarrow \mathbb{T}$, where \mathbb{X} is the statement space of language \mathcal{L} and \mathbb{T} is the tag space of OTTBE.

Setup $(1^\kappa, s, n)$:

Output $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]}) := \text{OTTBE.Setup}(1^\kappa, s, n)$.

Encrypt (pk, x, m) :

Output $c := \text{OTTBE.Encrypt}(\text{pk}, H(x), m)$.

ShareDec $(\text{pk}, c, x, w, (i, \text{sk}_i))$:

If $(x, w) \in R$, output $\text{OTTBE.ShareDec}(\text{pk}, c, H(x), (i, \text{sk}_i))$. Otherwise, output (i, \perp) .

ShareVf $(\text{pk}, \text{vk}, c, x, (i, d_i))$:

If $d_i = \perp$, output false. Otherwise output $\text{OTTBE.ShareVf}(\text{pk}, \text{vk}, c, H(x), (i, d_i))$.

Combine $(\text{pk}, \text{vk}, c, x, \{(i, d_i)\}_{i \in \mathcal{S}})$:

Output $\text{OTTBE.Combine}(\text{pk}, \text{vk}, c, H(x), \{(i, d_i)\}_{i \in \mathcal{S}})$.

Theorem 1. Let OTTBE be a threshold tag-based encryption scheme that is OB-IND-CCA secure and H be a collision-resistant hash function. Then, the scheme SO-TWE_{OTTBE} is a SO-IND-CCA secure threshold witness encryption scheme.

The security proof is presented in the full version of this paper [24].

Confidential witnesses and decryptions. We can add the support of confidential witnesses and decryptions to our construction by applying techniques from [8]. To ensure confidentiality of witnesses, clients send non-interactive zero-knowledge proofs of knowledge of the witness to the decryption servers. Then, as part of the decryption algorithm the servers check the validity of the proof against the submitted statement, instead of checking the witness relation directly. To achieve confidentiality of decryptions, the decryption servers encrypt

decryption shares under the public key of the client as part of the decryption algorithm. We ensure that decryption requests cannot be replayed with different public keys by applying the witness confidentiality approach and labeling the zero-knowledge proof with the submitted public key.

VI. O-TTBE FROM BILINEAR MAPPINGS AND RANDOM ORACLES

In this section, we present the construction of a concretely efficient oblivious threshold tag-based encryption scheme. Our construction is based on bilinear maps and random oracles and its security relies on the Decision Bilinear Diffie-Hellman assumption (cf. Section II-A). Before we present the formal specification of the construction, we give an intuition about the challenges of designing an O-TTBE scheme and how they are addressed by our construction.

Common approaches towards CCA security in the threshold setting incorporate ciphertext validation before decryption [18]–[22]. The validation ensures that each decryption request is either declined or yields exactly the original plaintext created by some client. This feature is the common way to prevent the adversary from executing *ciphertext-reuse*. Under this term, we understand reusing and potentially adapting ciphertext components in maliciously created ciphertext with the goal to extract decryptions for valid ciphertexts from the decryptions of maliciously created ones.

In an *oblivious threshold* scheme, declining decryptions is not possible since a single decryption server must not detect if the provided tag is valid. This is due to the fact that some servers can get corrupted in the threshold setting. If a single server was able to check the validity of a tag, the adversary would be able to exploit corrupted servers to break obliviousness. It follows that we have to apply a less strict ciphertext validation allowing for multiple decryptions, with different tags, of the same ciphertext. However, decryptions with invalid tags must not leak any information about the encrypted plaintext or the tag used for encryption. Consequently, we cannot follow the approaches of previous work.

Instead, we have to take one step back and address the challenge of achieving CCA security independent of previous work. It turns out that the discussed ciphertext validation is necessary but not sufficient to prove CCA security. In particular, when constructing a CCA secure encryption scheme it is not sufficient to take a CPA secure scheme and add a zero-knowledge proof of correct encryption to the ciphertexts. Proving security via a reduction to a number theoretic assumption is typically done by building a simulator that uses a concrete adversary on the scheme to break the underlying assumption. Even if a ciphertext is proven to be created correctly, the simulator needs to be capable of answering decryption queries of the adversary without actually knowing the secret key. This challenge is typically addressed by incorporating an additional trapdoor into the construction. It follows that for achieving CCA security we need both, (i) a way to prevent ciphertext reuse and (ii) a trapdoor to enable the simulator to answer decryption queries. In addition, for tag obliviousness, we

have to achieve the former while (iii) still allowing multiple decryptions, with different tags, for the same ciphertext.

We propose a new construction deploying a single extension together with a simple zero-knowledge proof of correct encryption to a standard threshold variant of CPA secure Hash-ElGamal. The extension provides both, (iii) tag obliviousness and (ii) a trapdoor for decryption, such that a simple zero-knowledge proof of correct encryption is sufficient to decline invalid ciphertexts, and hence, (i) prevent ciphertext reuse.

We start by briefly recalling Hash-ElGamal. In Hash-ElGamal, the encryption of a message m samples a random exponent a and outputs two elements $A = g^a$ and $M = m \oplus H(X^a)$ for a group generator g , a random oracle H , and a public key $X = g^x$. In the threshold setting, the secret key x is secret shared among the decryption servers. The decryption shares of the servers are calculated as $d_i := A^{x_i}$, where x_i is the share of the i -th server.

Our extension is to apply a random offset T to A for both, encryption and decryption, using a bilinear map e . This offset is unique for each ciphertext-tag pair. Precisely, we compute $M := H(e(T, X^a)) \oplus m$ for encryption and $d_i := e(T, A^{x_i})$ for decryption. As our notion requires each encrypting party and decryption server to be capable of generating the offset independently, we generate the offset using a random oracle. In particular, we compute the offset T by applying the random oracle to the tag t and the ciphertext component A , i.e., we compute $T = H_2(t, A)$. Finally, we add a non-interactive Schnorr zero-knowledge argument of knowledge of a [26] to the ciphertext, which we bind to the ciphertext component M . The binding is done by incorporating M into the generation of the challenge in the Fiat-Shamir transformation [27]. In addition, we add a Chaum-Pedersen zero-knowledge argument [28] of correct decryption to decryption shares.

The random offset T adds a random exponent $\log_g(T)$ to decryptions with invalid tags, and hence, ensures that invalid decryptions do not give any information about the encrypted message (cf. (iii)). Further it provides a backdoor that can be exploited by the simulator to answer decryption queries without knowledge of x_i (cf. (ii)). In particular, the simulator can simulate the random oracle such that the simulator learns $k = \log_g(T)$ for each T generated by H_2 . This way, the simulator can calculate the combined decryption shares $D = e(X^k, A) = e(T, A^x)$ which can again be used to interpolate decryption shares of individual parties. Finally, the zero-knowledge argument of knowledge of a ensures that a component A cannot be re-used for different ciphertexts, and hence, prevents ciphertext reuse (cf. (i)). Further, the zero-knowledge argument of correct decryption ensures that malicious servers cannot trick honest clients into accepting incorrect decryptions.

Our construction yields a concretely efficient scheme. The ciphertexts consist of a bitstring with length equal to the message length, a group element of the bilinear mapping's base group and two exponents (in \mathbb{Z}_q , where q is the bilinear group's order). Decryption shares consist of one group element in the mapping's target group and two exponents. Encryption

requires a single evaluation and decryption three evaluations of the bilinear map.

We continue by presenting the concrete construction:

Construction 2: $\text{TTBE}_{\text{pROM}}$

Public parameters:

The scheme is defined over a bilinear map $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ with groups \mathbb{G} and \mathbb{G}_T where each group is of order q . The number of committee members is denoted by n and the threshold parameter is s . The message and tag length is defined as l . We make use of random oracles $H_1 : \mathbb{G}_T \rightarrow \{0, 1\}^l$, $H_2 : \{0, 1\}^l \times \mathbb{G} \rightarrow \mathbb{G}$, $H_3 : \{0, 1\}^l \times \mathbb{G}^2 \rightarrow \mathbb{Z}_q$, $H_4 : \mathbb{G}^3 \rightarrow \mathbb{Z}_q$.

Setup($1^\kappa, s, n$):

Sample a generator, g , of \mathbb{G} , a secret key $x \in_R \mathbb{Z}_q$ and a sharing polynomial F of degree $s - 1$ over \mathbb{Z}_q such that $F(0) = x$. Set $\text{pk} = (g, X := g^x)$, $\text{vk} := \{g^{F(i)}\}_{i \in [n]}$ and $\text{sk}_i := F(i) = x_i$ for each $i \in [n]$. Output $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]})$.

Encrypt(pk, t, m):

Sample $a, r \in_R \mathbb{Z}_q$ and calculate:

$$A := g^a, T = H_2(t, A), \widetilde{M} := e(T, X^a), M := H_1(\widetilde{M}) \oplus m \\ U := g^r, w = H_3(M, A, U), f = r + aw, \pi := (w, f)$$

Return $c = (M, A, \pi)$.

Note that π constitutes a zero knowledge argument of knowledge of $\log_g(A)$.

ValidateCT(c):

Parse $c = (M, A, \pi = (w, f))$ and return true iff

$$w = H_3(M, A, U) \text{ for } U = \frac{g^f}{A^w}.$$

ShareDec($\text{pk}, c, t', (i, \text{sk}_i)$):

If **ValidateCT**(c) = false return (i, \perp) . Otherwise choose $r_i \in_R \mathbb{Z}_q$ and compute,

$$T' := H_2(t', A), D_i := e(T', A^{x_i}) \\ U_i := e(T', A^{r_i}), V_i := e(T', g^{r_i}) \\ w_i := H_4(D_i, U_i, V_i), f_i := r_i + x_i \cdot w_i \\ \pi_i := (w_i, f_i)$$

and return $d_i := (i, D_i, \pi_i)$.

Note that π_i constitutes a zero knowledge argument that $(e(T', A), e(T', \text{vk}_i), D_i)$ is a Diffie-Hellmann triple.

ShareVf($\text{pk}, \text{vk}_i, c, t', d_i$):

Parse $d_i = (i, D_i, \pi_i = (w_i, f_i))$, $c = (\cdot, A, \cdot)$, calculate $T' := H_2(t', A)$ and return true iff

$$w_i = H_4(D_i, U_i, V_i) \text{ for } U_i = \frac{e(T', A)^{f_i}}{D_i^{w_i}}, V_i = \frac{e(T', g)^{f_i}}{e(T', \text{vk}_i)^{w_i}}.$$

Combine($\text{pk}, \text{vk}, c, t', \{(d_i)\}_{i \in \mathcal{S}}$):

Return $m = M \oplus H_1(\prod_{i \in \mathcal{S}} (D_i)^{\lambda_{0,i}^{\mathcal{S}}})$.

Correctness of the scheme can be shown as follows:

$$m = M \oplus H_1\left(\prod_{i \in \mathcal{S}} (D_i)^{\lambda_{0,i}^{\mathcal{S}}}\right) = M \oplus H_1\left(\prod_{i \in \mathcal{S}} (e(T', A^{x_i}))^{\lambda_{0,i}^{\mathcal{S}}}\right) \\ = M \oplus H_1\left(\prod_{i \in \mathcal{S}} e(T', A)^{x_i \cdot \lambda_{0,i}^{\mathcal{S}}}\right) = M \oplus H_1(e(T', A)^x) \\ = m \oplus H_1(e(T, g^{ax})) \oplus H_1(e(T', g^{ax})) = m,$$

where $t = t'$ yields $H_2(t, A) = T = T' = H_2(t', A)$.

For security, we state the following theorem:

Theorem 2. Let BGen be a Bilinear Group Generator, $(e, \mathbb{G}, \mathbb{G}_T, q) \leftarrow_R \text{BGen}(\kappa)$ be a bilinear group in which the Decisional Bilinear Diffie-Hellman (DBDH) assumption holds, and H_1, H_2, H_3, H_4 be programmable random oracles. Then, the scheme $\text{TTBE}_{\text{pROM}}$ is an OB-IND-CCA secure oblivious threshold tag-based encryption scheme.

We will provide an intuition of our proof for indistinguishable messages under chosen-ciphertext attacks, here, and defer the formal security proof for both indistinguishable messages under chosen-ciphertext attacks (defined via $\text{Exp}_{\text{TTBE}_{\text{pROM}}, \mathcal{A}}^{\text{O-CCA}}$) and decryption consistency under chosen-ciphertext attacks (defined via $\text{Exp}_{\text{TTBE}_{\text{pROM}}, \mathcal{A}}^{\text{O-DC}}$) to the full version of this paper [24].

Proof intuition. We prove indistinguishable messages via a reduction to the DBDH assumption. Hence, we build a distinguisher \mathcal{D} that receives a tuple $(\bar{g}, \bar{h}, \alpha = \bar{g}^x, \beta = \bar{g}^y, \gamma)$ and decides if the received tuple is a DBDH tuple, i.e., if $\gamma = e(\bar{h}, \bar{g})^{xy}$. \mathcal{D} has access to an adversary \mathcal{A} on the experiment $\text{Exp}_{\text{TTBE}_{\text{pROM}}, \mathcal{A}}^{\text{O-CCA}}$.

The reduction is based on the observation that, in order to win in $\text{Exp}_{\text{TTBE}_{\text{pROM}}, \mathcal{A}}^{\text{O-CCA}}$, adversary \mathcal{A} when receiving a challenge ciphertext (M^*, A^*, \cdot) has to query H_1 at either $P_0 = e(H_2(t_0, A^*), A^*)^x$ or $P_1 = e(H_2(t_1, A^*), A^*)^x$. If \mathcal{D} defines public parameters $g = \bar{g}$ and $\text{pk} = X = \alpha$ and challenge ciphertext components $H_2(t_{1-b}, A^*) \leftarrow \bar{h}$ (via programming of the random oracle) and $A^* = \beta$ for some $b \in_R \{0, 1\}$, it follows that $P_{1-b} = \gamma$ iff the received tuple is a DBDH tuple. Hence, we can distinguish DBDH tuples from random tuples based on the event that γ has been queried by \mathcal{A} . However, setting $M^* = m_{b_m} \oplus H_1(\gamma)$ for $b_m \in_R \{0, 1\}$ does not yield a valid ciphertext if the tuple is no DBDH tuple, a fact that makes the reduction distinguishable from a real experiment. While there are techniques to deal with this problem (cf. [19]), this distinguishability makes the argumentation more long-winded. Instead, we make use of the fact that we are in the tag-based setting, i.e., there are two possible keys at which H_1 can be queried to decide which tag or message has been used for encryption. In particular, we create M^* such that it is a correct encryption of m_{b_m} under tag t_b , i.e., $M^* = m \oplus H_1(P_b)$. At the same time, we program H_2 such that $P_{1-b} = \gamma$ iff the received tuple is a DBDH tuple, i.e., by programming $H_2(t_{1-b}, A^*) \leftarrow \bar{h}$. Hence, we can distinguish based on the event that γ has been queried while still creating a valid ciphertext.

The next question is how to actually compute P_b without knowing x nor $y = \log_g(A^*)$. Here we make use of the fact that \mathcal{D} simulates the random oracle H_2 that is used to compute the tag-dependent offset. In particular, whenever the random oracle H_2 is supposed to sample a random value in \mathbb{G} , it samples a random exponent $k \in_R \mathbb{Z}_q$ instead and returns \bar{g}^k . The output is still uniformly random distributed in \mathbb{G} but \mathcal{D} learns the discrete logarithm of every value sampled by H_2 . This way, \mathcal{D} can restore $k = \log_g(H_2(t_b, A^*))$ and compute $P_b = e(\alpha, \beta^k) = e(H_2(t_b, A^*), A^*)^x$.

As explained above, the major challenge is to answer decryption queries without having access to the private key x . However, this problem can be solved the same way as computing P_b . In particular, \mathcal{D} answers decryption queries for ciphertext $c = (\cdot, A, \cdot)$ and tag t by restoring $k = \log_g(H_2(t, A))$ and computing $e(\beta, A^k) = e(H_2(t, A), A^x)$. The only keys for which the restoring of the exponent k does not work are $(t_{1-b}, A^*) = (t_{1-b}, \beta)$ for which \mathcal{D} programmed the random oracle to \bar{h} without knowing $\log_g(\bar{h})$. However, in consistency with the original security game, \mathcal{D} declines decryptions for (c, t) if $(c, t) \in \{(c^*, t_0), (c^*, t_1)\}$. Hence, \mathcal{D} only fails to answer decryption queries if \mathcal{A} sends a valid ciphertext $c \neq c^*$ such that $c = (\cdot, A^*, \pi)$. However, to do so, the adversary needs to be capable of generating a valid zero-knowledge argument π of knowledge of $y = \log_g(A^*)$ without actually knowing y . In fact, not even \mathcal{D} has knowledge of y . The probability is negligible for a computationally bounded adversary to find such a proof. It follows that \mathcal{D} is capable of answering all decryption queries, except with negligible probability.

VII. OBLIVIOUS TTBE FROM ANONYMOUS TIBE

This section presents a general transformation from an anonymous threshold identity-based encryption scheme, a one-time signature scheme and a collision-resistant hash functions to an oblivious threshold tag-based encryption scheme. The scheme depicts an extension of [18].

Construction 3: TTBE_{IBE}

Public parameters:

The number of committee members is denoted by n and the threshold parameter is s . We make use of a one-time signature scheme OTS, an anonymous threshold identity-based encryption scheme TIBE, and a collision-resistant hash function $H : \mathbb{T} \times \mathbb{K} \rightarrow \mathbb{I}$, where $\mathbb{T}, \mathbb{K}, \mathbb{I}$ is the tag space, the verification key space of OTS, and the identity space of TIBE.

Setup($1^\kappa, s, n$):

Run $(pk, vk, \{sk_i\}_{i \in [n]}) \leftarrow \text{TIBE.Setup}(1^\kappa, s, n)$ and output the keys $(pk, vk, \{sk_i\}_{i \in [n]})$.

Encrypt(pk, t, m):

Generate a signature key pair $(\text{SigK}, \text{VerK}) \leftarrow \text{OTS.KeyGen}(1^\kappa)$, calculate $\text{id} := H(t, \text{VerK})$, $c_0 := \text{TIBE.Encrypt}(pk, \text{id}, m)$, and $\sigma := \text{OTS.Sign}(\text{SigK}, c_0)$. Output $c := (c_0, \text{VerK}, \sigma)$.

ShareDec($pk, c, t, (i, sk_i)$):

Parse c to $(c_0, \text{VerK}, \sigma)$ and check that $\text{OTS.Verify}(\text{VerK}, \sigma, c_0) = \text{true}$. If the check fails, output (i, \perp) . Otherwise, calculate $\text{id} := H(t, \text{VerK})$ and output an identity key share $(i, ik_i) \leftarrow \text{TIBE.ShareKeyGen}(pk, i, sk_i, \text{id})$ as d_i .

ShareVf($pk, vk, c, t, (i, d_i)$):

Parse c to $(c_0, \text{VerK}, \sigma)$ and output true iff $\text{OTS.Verify}(\text{VerK}, \sigma, c_0) = \text{true}$ and $\text{TIBE.ShareVf}(pk, vk, H(t, \text{VerK}), i, d_i) = \text{true}$.

Combine($pk, vk, c, t, \{(i, d_i)\}_{i \in \mathcal{S}}$):

Parse c to $(c_0, \text{VerK}, \sigma)$, calculate $\text{id} = H(t, \text{VerK})$ and $ik := \text{TIBE.Combine}(pk, vk, \text{id}, \{(d_i)\}_{i \in \mathcal{S}})$. If $ik = \perp$, output \perp .

Otherwise, output $m := \text{TIBE.Decrypt}(pk, \text{id}, ik, c_0)$.

Correctness of the scheme is easy to see. If the same tag is used for decryption and encryption, the encryption contains a ciphertext under the same identity for which the decryption algorithm creates the identity key. Next, we show security.

Theorem 3. *Let TIBE be an anonymous threshold identity-based encryption scheme that is ANON-IND-ID-CPA secure, H be collision-resistant hash function, and OTS an existentially unforgeable one-time signature scheme. Then, the scheme TTBE_{IBE} is a OB-IND-CCA secure threshold tag-based encryption scheme.*

The security proof is presented in the full version of the paper [24].

VIII. CONSTRUCTING ANONYMOUS TIBE

In this section, we construct an anonymous threshold identity-based encryption scheme (TIBE) from an anonymous non-threshold verifiable identity-based encryption scheme (VIBE) and an homomorphic secret sharing scheme (HSS) with linear decoding. A VIBE extends the definition of an identity-based encryption scheme with a verification algorithm that allows to check if an identity key was generated correctly. An HSS scheme allows to secret share some value and to perform operations on the shares such that the result of the combination yields the output of a function applied directly to the value. We state the definitions for VIBE and HSS in Appendix B-C and B-D respectively. The HSS scheme is used to execute the Extract algorithm of the VIBE scheme in a distributed way. The operations that need to be supported by the HSS scheme depend on the concrete VIBE scheme, i.e., how the output shares of its Extract algorithm can be computed. While we use the HSS scheme in a black-box way, it is an interesting open question to provide concrete instantiations of the following black-box transformation. In the full version of this paper [24], we discuss potential pathways to obtain an anonymous threshold IBE scheme from the concrete anonymous IBE scheme by Boyen and Waters [16].

Construction 4: Anonymous TIBE

Public parameters:

The number of committee members is denoted by n and the threshold parameter is s . This construction uses an ANON-IND-ID-CPA secure VIBE scheme $\text{VIBE} = (\text{VIBE.Setup}, \text{VIBE.Extract}, \text{VIBE.Verify}, \text{VIBE.Encrypt}, \text{VIBE.Decrypt})$ and a linear decoding HSS scheme $\text{HSS} = (\text{HSS.Share}, \text{HSS.Eval}, \text{HSS.Dec})$ for the function $y := (ik_z, \rho_z) \leftarrow \text{VIBE.Extract}(pk, x, z)$ with public input $z = \text{id}$ and shared private input $x = \text{msk}$, as building blocks.

Setup($1^\kappa, s, n$):

- $(pk_{\text{VIBE}}, vk_{\text{VIBE}}, \text{msk}) \leftarrow \text{VIBE.Setup}(1^\kappa)$
- $(\text{msk}_1, \dots, \text{msk}_n) \leftarrow \text{HSS.Share}(1^\kappa, \text{msk})$
- **return** $(pk, vk, (sk_1, \dots, sk_n))$:=
 $(pk_{\text{VIBE}}, vk_{\text{VIBE}}, (\text{msk}_1, \dots, \text{msk}_n))$

ShareKeyGen(pk, i, sk_i, id):

- **return** (i, ik_i) , where $ik_i := y_i \leftarrow \text{HSS.Eval}(i, \text{id}, sk_i)$

ShareVf(pk, vk, id, i , ik_i):

- **return true**

Combine(pk, vk, id, $\{(i, ik_i)\}_{i \in \mathcal{S}}$):

- $y \leftarrow \text{HSS.Dec}(\{ik_i\}_{i \in \mathcal{S}})$
- Parse $y := (ik, \rho)$
- **if** VIBE.Verify(pk, vk, id, ik, ρ) = 1 **return** ik
- **else return** \perp

Encrypt(pk, id, m):

- **return** VIBE.Encrypt(pk, id, m)

Decrypt(pk, id, ik, c):

- **return** VIBE.Decrypt(pk, ik, c)

We first show that our construction satisfies the correctness properties, in particular share consistency and decryption correctness. Then, we prove the security property, ANON-IND-ID-CPA security.

The share consistency property states that for all correctly generated identity key shares, the ShareVf algorithm outputs true. Since the ShareVf algorithm of our construction always outputs true, the property is apparently satisfied. Decryption correctness is easy to see as well. Let, for any $\kappa, n \in \mathbb{N}$ and $1 \leq s \leq n$, $(pk, vk, \{sk_i\}_{i \in [n]}) \leftarrow \text{Setup}(1^\kappa, s, n)$. Note that $sk_i := msk_i$ where msk_i is the i -th share obtained using $\text{HSS.Share}(1^\kappa, msk)$ for a master secret key of the non-threshold VIBE scheme VIBE. Then, for any id, $\text{ShareKeyGen}(pk, i, sk_i, \text{id})$ returns an output share of $\text{HSS.Eval}(i, \text{id}, msk_i)$ which equals a share of $\text{VIBE.Extract}(pk, msk, \text{id})$. Given any set of s identity key shares, the Combine algorithm first decodes the shares to (ik, ρ) and outputs ik if $\text{VIBE.Verify}(pk, vk, \text{id}, ik, \rho) = 1$. Due to the correctness property of the (s, n) -HSS scheme, (ik, ρ) is exactly the output of $\text{VIBE.Extract}(pk, msk, \text{id})$. Now, due to the correctness property of VIBE, it follows that $\text{Decrypt}(pk, ik, \text{Encrypt}(pk, \text{id}, m)) = m$ holds for any message m .

Finally, we show that the scheme TIBE is ANON-IND-ID-CPA secure. Formally, we state the following theorem.

Theorem 4. *Let VIBE be an ANON-IND-ID-CPA secure VIBE scheme satisfying soundness and let HSS be a linear decoding (s, n) -HSS scheme satisfying correctness and computational security. Then, TIBE defined in Construction 4 is an ANON-IND-ID-CPA secure (n, s) -TIBE scheme.*

The security proof is presented in the full version of the paper [24].

IX. APPLICATIONS

Statement-oblivious threshold witness encryption (SO-TWE) is interesting whenever use-cases of classical witness encryption, e.g., the ones presented in [1], [8], should be extended by an additional privacy property, i.e., if the statement used for encryption is required to stay hidden until

the decryption is successful. A straightforward example is witness encryption based time-lock encryption, as proposed by [6], with a hidden release time. In simplified settings, in which the decryption servers have access to public data (e.g., timestamps), our intermediate notion of oblivious threshold tag-based encryption (O-TTBE) is often sufficient. However, in more sophisticated scenarios, e.g., if the decryption servers need to rely on external authorities to provide authenticated public data, it is necessary to use SO-TWE. We present use-cases and briefly explain how they can be realized using our primitives; one of the use-case is provided in this section and others are deferred to the full version of this paper [24]. The use-cases are partial extensions to the ones presented by [8] for their notion of eWEB.

Price-dependent transaction execution with hidden price. Imaging a user that wants to buy some asset at a Decentralized Finance (DeFi) trading platform once the share price reaches a certain value. Since the user does not know when this event happens, it does not want to stay online all the time. The user's goal is to keep the transaction and the desired share price private until the price hits the intended value. Privacy is an important aspect in this scenario, since revealing information, e.g., the intended purchase price, could lead to financial disadvantage, e.g., due to insider trading. In the DeFi space, oracle services are widely deployed and commonly used. These services provide signed information about real-world data such as share prices. However, achieving the user's goal requires additional techniques. To support the described scenario, the user can exploit SO-TWE.

In more detail, suppose there is a committee holding the secret key shares of a SO-TWE scheme with public key pk for language \mathcal{L} with associated relation R . \mathcal{L} is defined such that a statement x specifies the intended share price as well as the public key of the oracle service and $(x, w) \in R$ if the witness w contains a proof that the current share price equals the specified one signed by the oracle. Initially, the user creates a transaction tx containing the trade description and encrypts it using the public key of the SO-TWE scheme, i.e., $c = \text{Encrypt}(pk, x, \text{tx})$, where the statement is from the specified language. The user sends the ciphertext c to its broker. Next, the broker regularly requests the current share price together with a proof from the oracle and provides this information as the witness w together with the ciphertext c to the decryption committee. Each committee member performs $\text{ShareDec}(pk, c, x, w, (i, sk_i))$ to obtain a decryption share (i, d_i) . After obtaining s valid shares from the committee, the broker executes $\text{Combine}(pk, vk, c, x, \{(i, d_i)\}_{i \in \mathcal{S}})$. If decryption was executed with the intended share price, the result is tx. In this case, the broker executes the transaction which effectively performs the trade. Otherwise, the output of the Combine-algorithm does not constitute a valid transaction.

The statement-obliviousness property guarantees that no party, not even the broker or the oracles, gets to know anything about the trade, neither the asset, the amount or the specified price, until the transaction is successfully decrypted and the trade can be executed. This way, we prevent insider trading. To

incentivize the broker to execute the trade reliably and timely, users can rely on multiple brokers rewarding the one executing the trade first.

ACKNOWLEDGMENTS

The first, third, and fourth authors were supported by the German Federal Ministry of Education and Research (BMBF) *iBlockchain project* (grant nr. 16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 – 236615297 (CROSSING Project S7)*, and by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*. The second author was supported by ISF grant No. 1316/18 and by the Algorand Centres of Excellence programme managed by Algorand Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Algorand Foundation.

REFERENCES

- [1] S. Garg, C. Gentry, A. Sahai, and B. Waters, “Witness encryption and its applications,” in *STOC*, 2013.
- [2] A. R. Choudhuri, M. Green, A. Jain, G. Kaptchuk, and I. Miers, “Fairness in an unfair world: Fair multiparty computation from public bulletin boards,” in *CCS*, 2017.
- [3] S. Goldwasser, Y. T. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, “How to run turing machines on encrypted data,” in *CRYPTO*, 2013.
- [4] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, “Candidate indistinguishability obfuscation and functional encryption for all circuits,” in *FOCS*, 2013.
- [5] C. Gentry, A. B. Lewko, and B. Waters, “Witness encryption from instance independent assumptions,” in *CRYPTO*, 2014.
- [6] J. Liu, T. Jager, S. A. Kakvi, and B. Warinschi, “How to build time-lock encryption,” *Des. Codes Cryptogr.*, 2018.
- [7] D. Boneh, D. B. Glass, D. Krashen, K. E. Lauter, S. Sharif, A. Silverberg, M. Tibouchi, and M. Zhandry, “Multiparty non-interactive key exchange and more from isogenies on elliptic curves,” *J. Math. Cryptol.*, 2020.
- [8] V. Goyal, A. Kothapalli, E. Masserova, B. Parno, and Y. Song, “Fast batched dpss and its applications,” in *PKC*, 2022.
- [9] A. Erwig, S. Faust, and S. Riahi, “Large-scale non-interactive threshold cryptosystems through anonymity,” *IACR Cryptol. ePrint Arch.*, 2021.
- [10] M. Campanelli, B. David, H. Khoshakhlagh, A. K. Kristensen, and J. B. Nielsen, “Encryption to the future: A paradigm for sending secret messages to future (anonymous) committees,” *IACR Cryptol. ePrint Arch.*, 2021.
- [11] V. Goyal, E. Masserova, B. Parno, and Y. Song, “Blockchains enable non-interactive MPC,” in *TCC*, 2021.
- [12] G. Almashaqbeh, F. Benhamouda, S. Han, D. Jaroslawicz, T. Malkin, A. Nicita, T. Rabin, A. Shah, and E. Tromer, “Gage MPC: bypassing residual function leakage for non-interactive MPC,” *Proc. Priv. Enhancing Technol.*, 2021.
- [13] F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin, “Can a public blockchain keep a secret?” in *TCC*, 2020.
- [14] C. Gentry, S. Halevi, B. Magri, J. B. Nielsen, and S. Yakoubov, “Random-index PIR and applications,” in *TCC*, 2021.
- [15] S. Arita and K. Tsurdome, “Construction of threshold public-key encryptions through tag-based encryptions,” in *ACNS*, 2009.
- [16] X. Boyen and B. Waters, “Anonymous hierarchical identity-based encryption (without random oracles),” in *CRYPTO*, 2006.
- [17] C. Gentry, “Practical identity-based encryption without random oracles,” in *EUROCRYPT*, 2006.
- [18] D. Boneh, X. Boyen, and S. Halevi, “Chosen ciphertext secure public key threshold encryption without random oracles,” in *CT-RSA*, 2006.

- [19] V. Shoup and R. Gennaro, “Securing threshold cryptosystems against chosen ciphertext attack,” in *EUROCRYPT*, 1998.
- [20] P. Mol and S. Yilek, “Chosen-ciphertext security from slightly lossy trapdoor functions,” in *PKC*, 2010.
- [21] B. Libert and M. Yung, “Non-interactive cca-secure threshold cryptosystems with adaptive security: New framework and constructions,” in *TCC*, 2012.
- [22] V. Koppula and B. Waters, “Realizing chosen ciphertext security generically in attribute-based encryption and predicate encryption,” in *CRYPTO*, 2019.
- [23] T. E. Gamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” in *CRYPTO*, 1984.
- [24] S. Faust, C. Hazay, D. Kretzler, and B. Schlosser, “Statement-oblivious threshold witness encryption,” Cryptology ePrint Archive, Paper 2023/668, 2023, <https://eprint.iacr.org/2023/668>. [Online]. Available: <https://eprint.iacr.org/2023/668>
- [25] D. Boneh and M. K. Franklin, “Identity-based encryption from the weil pairing,” in *CRYPTO*, 2001.
- [26] C. Schnorr, “Efficient identification and signatures for smart cards,” in *CRYPTO*, 1989.
- [27] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems,” in *CRYPTO*, 1986.
- [28] D. Chaum and T. P. Pedersen, “Wallet databases with observers,” in *CRYPTO*, 1992.
- [29] E. Boyle, N. Gilboa, Y. Ishai, H. Lin, and S. Tessaro, “Foundations of homomorphic secret sharing,” in *ITCS*, 2018.

APPENDIX A

REDUCTION: EXTRACTABILITY TO SO-IND-CCA SECURITY

This section provides further details to the notion of extractable threshold witness encryption and presents the reduction from extractability to SO-IND-CCA security in the threshold setting.

Intuitively, the original notion of extractable witness encryption states that any adversary that is able to obtain non-trivial information about a plaintext is also able to provide the witness for the corresponding ciphertext. Formally, this is defined by allowing the adversary to win the security game with non-negligible advantage but requiring that such an adversary can be used to extract the witness for the challenged plaintext. It is natural to translate this notion from witness encryption to our context, the one of threshold witness encryption, by defining extractability via the same security game as the one of SO-IND-CCA security, $\text{Exp}^{\text{SO-CCA}}$, with the only difference that the adversary is allowed to query the decryption oracle $\mathcal{O}(\cdot, \cdot, \cdot, \cdot, \cdot)$ with any ciphertext-witness pair while the oracle in the original experiment returns \perp if $c = c^*$, $x \in \{x_0, x_1\}$ and $(x, (w_s, w_p)) \in R$. We call this game $\text{Exp}_{\text{TWE}, \mathcal{A}}^{\text{SO-Ext}}$, when played with an adversary \mathcal{A} for a scheme TWE.

Extractability now requires that if the adversary has a non-negligible advantage in the security game, then it is possible to construct a witness extractor that extracts a valid witness with non-negligible probability.

Definition 7 (Extractability of SO-TWE). *Let \mathcal{A} be a PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$ such that the following holds: for every pk generated by Setup, for every x_0, x_1, m_0, m_1 and every auxiliary information $z \in \{0, 1\}^{\text{poly}(\kappa)}$:*

$$\Pr[\text{Exp}_{\text{TWE}, \mathcal{A}}^{\text{SO-Ext}}(1^\kappa) = 1] \geq \frac{1}{4} + \frac{1}{\text{poly}(\kappa)}.$$

Then there exists a PPT extractor \mathcal{E} such that:

$$\Pr[(b, w) = \mathcal{E}(1^\kappa, x_0, x_1, z) : (x_b, w) \in R] \geq \frac{1}{\text{poly}(\kappa)}.$$

We state the following theorem:

Theorem 5. Any statement-oblivious threshold witness encryption scheme TWE, that is SO-IND-CCA secure, is also extractable.

Proof: Assume an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$ that breaks extractability of TWE. This means that \mathcal{A} is able to win game $\text{Exp}_{\text{TWE}, \mathcal{A}}^{\text{SO-Ext}}(1^\kappa)$ with a non-negligible advantage and there is no extractor \mathcal{E} .

From the fact that there is no extractor, we can derive that \mathcal{A} does not query the oracle with input $(\cdot, \cdot, x_b, w_s, w_p)$ such that $x_b \in \{x_0, x_1\}$ and $R(x_b, (w_s, w_p)) = \text{true}$; otherwise, there would exist the trivial extractor $\mathcal{E}_{\text{triv}}$ that equals \mathcal{A} up to this query and then outputs $(x_b, (w_s, w_p))$.

As the adversary \mathcal{A} does not make such queries, it can be used in the SO-IND-CCA game without any modifications and will still win with non-negligible probability. ■

APPENDIX B FURTHER DEFINITIONS

A. Collision Resistance of Hash Functions

The collision resistance property of hash functions states that any PPT adversary can find two values x, x' such that $x \neq x'$ and $H(x) = H(x')$ only with negligible probability.

B. Security Properties of Digital Signatures

We assume digital signatures to satisfy consistency and existential unforgeability against chosen-message attacks. The consistency property states that for all $\kappa \in \mathbb{N}$, for all $(\text{SigK}, \text{VerK}) \leftarrow \text{KeyGen}(1^\kappa)$ and for every $m \in \mathbb{M}$ it holds $\Pr[\text{Verify}(\text{VerK}, m, \text{Sign}(\text{SigK}, m))] = 1$.

We define existential unforgeability against chosen-message attacks via the following game

Experiment $\text{Exp}_{\text{SIG}, \mathcal{A}}^{\text{EX-UNF}}(\kappa)$ <hr/> $(\text{SigK}, \text{VerK}) \leftarrow \text{KeyGen}(1^\kappa)$ $(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathcal{O}(\cdot)}(\text{VerK})$ if $\text{Verify}(\text{VerK}, m^*, \sigma^*) = 1$ then return 1 else return 0
--

where the adversary may ask its oracle \mathcal{O} on a message $m \in \mathbb{M}$ and gets back the signature $\sigma \leftarrow \text{Sign}(\text{SigK}, m)$. The pair (m^*, σ^*) output by \mathcal{A} must be different to any (m, σ) obtained by the oracle.

Definition 8. A signature scheme SIG is existentially unforgeable against chosen-message attacks if for every $\kappa \in \mathbb{N}$ and every PPT adversary \mathcal{A} there exists a negligible function negl such that

$$\Pr[\text{Exp}_{\text{SIG}, \mathcal{A}}^{\text{EX-UNF}}(\kappa) = 1] \leq \text{negl}(\kappa).$$

Definition 9 (OTS). A signature scheme OTS = (KeyGen, Sign, Verify) is called one-time signature scheme

with existential unforgeability against chosen-message attacks, if for every $\kappa \in \mathbb{N}$ and every PPT adversary \mathcal{A}' that makes at most one oracle query there exists a negligible function negl such that

$$\Pr[\text{Exp}_{\text{OTS}, \mathcal{A}'}^{\text{EX-UNF}}(\kappa) = 1] \leq \text{negl}(\kappa).$$

C. Verifiable IBE

We state a definition for verifiable identity-based encryption as an extension of identity-based encryption presented by Boneh and Franklin [25]. In particular, the primitive contains a verification algorithm that allows to check if an identity key is generated correctly.

Definition 10 (VIBE). A verifiable identity-based encryption scheme VIBE consists of five probabilistic polynomial-time algorithms:

- 1) $\text{Setup}(1^\kappa)$ takes as input a security parameter 1^κ and outputs a public key pk including public parameters, a verification key vk and a master key msk . pk include a description of the finite message space \mathbb{M} and the finite ciphertext space \mathbb{C} .
- 2) $\text{Extract}(\text{pk}, \text{msk}, \text{id})$ takes as input the public key pk , the master key msk and an identity $\text{id} \in \{0, 1\}^*$. It outputs an identity secret key ik_{id} together with a proof ρ_{id} stating that ik_{id} was computed correctly.
- 3) $\text{Verify}(\text{pk}, \text{vk}, \text{id}, \text{ik}_{\text{id}}, \rho_{\text{id}})$ takes as input the public key pk , the verification key vk , an identity id , an identity key ik_{id} , and a proof ρ_{id} . It outputs 1 if ik_{id} is a valid identity key for identity id and 0 otherwise.
- 4) $\text{Encrypt}(\text{pk}, \text{id}, m)$ takes as input the public key pk , an identity id and a message m . It outputs a ciphertext c encrypted under identity id .
- 5) $\text{Decrypt}(\text{pk}, \text{ik}_{\text{id}}, c)$ takes as input the public key pk , an identity secret key ik_{id} and a ciphertext c . It outputs a message m .

We require these algorithms to fulfill the following correctness and verifiability properties for all $\kappa \in \mathbb{N}$:

- **Correctness:** For every $(\text{pk}, \text{vk}, \text{msk}) \leftarrow \text{Setup}(1^\kappa)$, every $\text{id} \in \{0, 1\}^*$, every $(\text{ik}_{\text{id}}, \cdot) \leftarrow \text{Extract}(\text{pk}, \text{msk}, \text{id})$ and every $m \in \mathbb{M}$:

$$\text{Decrypt}(\text{pk}, \text{ik}_{\text{id}}, \text{Encrypt}(\text{pk}, \text{id}, m)) = m.$$

- **Verifiability:** For every $(\text{pk}, \text{vk}, \text{msk}) \leftarrow \text{Setup}(1^\kappa)$, every $\text{id} \in \{0, 1\}^*$ and every $(\text{ik}_{\text{id}}, \rho_{\text{id}}) \leftarrow \text{Extract}(\text{pk}, \text{msk}, \text{id})$

$$\text{Verify}(\text{pk}, \text{vk}, \text{id}, \text{ik}_{\text{id}}, \rho_{\text{id}}) = 1.$$

We define security by three properties: soundness, anonymity and security against chosen-plaintext attacks. We start defining the soundness property. Informally, soundness means that an adversary cannot come up with two different but valid identity keys that decrypt a chosen ciphertext to two different plaintexts. Formally, we define the soundness property via the following game.

Experiment $\text{Exp}_{\text{VIBE},\mathcal{A}}^{\text{SOUND}}(\kappa)$ <hr/> $(\text{pk}, \text{vk}, \text{msk}) \leftarrow \text{Setup}(1^\kappa)$ $(\text{ID}, c, (\text{ik}_{\text{ID}}, \rho_{\text{ID}}), (\text{ik}'_{\text{ID}}, \rho'_{\text{ID}})) \leftarrow \mathcal{A}^{\mathcal{O}(\cdot)}(\text{pk}, \text{vk})$ if $\text{Verify}(\text{pk}, \text{vk}, \text{ID}, \text{ik}_{\text{ID}}, \rho_{\text{ID}}) = 1$ $\wedge \text{Verify}(\text{pk}, \text{vk}, \text{ID}, \text{ik}'_{\text{ID}}, \rho'_{\text{ID}}) = 1$ $\wedge \text{Decrypt}(\text{pk}, \text{ik}_{\text{ID}}, c) \neq \text{Decrypt}(\text{pk}, \text{ik}'_{\text{ID}}, c)$ return 1 else return 0
--

The adversary can use its oracle $\mathcal{O}(\cdot)$ to make identity key queries. More precisely, upon receiving id from \mathcal{A} the oracle returns $(\text{ik}_{\text{id}}, \rho_{\text{id}}) \leftarrow \text{Extract}(\text{pk}, \text{msk}, \text{id})$ for any $\text{id} \in \{0, 1\}^*$.

Definition 11 (Soundness). *A verifiable identity-based encryption scheme VIBE satisfies soundness if for all $\kappa \in \mathbb{N}$ and all PPT adversary*

$$\Pr[\text{Exp}_{\text{VIBE},\mathcal{A}}^{\text{SOUND}}(\kappa) = 1] \leq \text{negl}(\kappa).$$

We next move on to the anonymity and security against chosen-plaintext attacks. The anonymity property of a VIBE scheme informally states that an adversary cannot learn the associated identity from a ciphertext, while the security against chosen-plaintext attacks states that an adversary cannot distinguish two ciphertexts over different messages. We combine both properties following Gentry [17] and define security via the following game.

Experiment $\text{Exp}_{\text{VIBE},\mathcal{A}}^{\text{A-V-CPA}}(\kappa)$ <hr/> $(\text{pk}, \text{vk}, \text{msk}) \leftarrow \text{Setup}(1^\kappa)$ $(\text{ID}_0, \text{ID}_1, m_0, m_1) \leftarrow \mathcal{A}_0^{\mathcal{O}}(\text{pk}, \text{vk})$ $\alpha, \beta \in_R \{0, 1\}$ $c^* \leftarrow \text{Encrypt}(\text{pk}, \text{ID}_\alpha, m_\beta)$ $(\alpha', \beta') \leftarrow \mathcal{A}_1^{\mathcal{O}}(c^*)$ return $(\alpha, \beta) = (\alpha', \beta')$
--

In the game $\text{Exp}_{\text{VIBE},\mathcal{A}}^{\text{A-V-CPA}}$, the adversary can use its oracle \mathcal{O} to make key generation queries. Upon receiving id , \mathcal{O} returns $\text{Extract}(\text{pk}, \text{msk}, \text{id})$ if $\text{id} \notin \{\text{ID}_0, \text{ID}_1\}$ and \perp otherwise.

Definition 12 (ANON-IND-ID-CPA). *A VIBE scheme VIBE is ANON-IND-ID-CPA secure if for all PPT adversary \mathcal{A} in game $\text{Exp}_{\text{VIBE},\mathcal{A}}^{\text{A-V-CPA}}$, there exists a negligible function negl such that*

$$\left| \Pr[\text{Exp}_{\text{VIBE},\mathcal{A}}^{\text{A-V-CPA}}(\kappa) = 1] - \frac{1}{4} \right| \leq \text{negl}(\kappa).$$

In the full version of this paper [24], we show how to construct a VIBE scheme from a standard identity-based encryption scheme IBE combined with an existentially unforgeable signature scheme SIG. Assuming IBE satisfies ANON-IND-ID-CPA security, the VIBE construction satisfies soundness and ANON-IND-ID-CPA security.

D. Homomorphic Secret Sharing

We follow the definition of Boyle et al. [29] for homomorphic secret sharing (HSS) schemes but state a simplified version that fits our application. In particular, we consider only a single input HSS and incorporate robust decoding in our definition where only s output shares are required for correct decoding. Additionally, we use the notation of s -out-of- n HSS to denote an n -server $(s-1)$ -secure HSS according to the definition of Boyle et al.

In Section VIII, we utilize an HSS to transform a VIBE scheme into a threshold IBE scheme. In particular, the identity key generation will be executed in a distributed fashion, i.e., the Extract algorithm of the non-threshold scheme. The homomorphic operations that need to be supported by the HSS depend on the concrete VIBE construction. Since we present a black-box construction in Section VIII, we consider a generalized homomorphic secret sharing scheme.

Definition 13 (HSS). *An s -out-of- n homomorphic secret sharing scheme HSS for a function $F : (\{0, 1\}^*)^2 \rightarrow \{0, 1\}^*$, or (s, n) -HSS in short, consists of three PPT algorithms:*

- 1) $\text{Share}(1^\kappa, x)$ takes as input a security parameter 1^κ and a user input x . It outputs n shares (x_1, \dots, x_n) , where server i gets share x_i .
- 2) $\text{Eval}(i, z, x_i)$ takes as input a server index i , a public input z and the i -th share x_i . It outputs $y_i \in \{0, 1\}^*$, corresponding to server i 's share of $F(z; x)$.
- 3) $\text{Dec}(\{y_i\}_{i \in \mathcal{S}})$ takes as input a set of output shares and outputs the final output $y \in \{0, 1\}^*$.

We require the following correctness and security properties for every $\kappa \in \mathbb{N}$:

- **Correctness:** For any input $z, x \in \{0, 1\}^*$ and any set of shares $(x_1, \dots, x_n) \leftarrow \text{Share}(1^\kappa, x)$. Let $\forall i \in [n]$ $y_i \leftarrow \text{Eval}(i, z, x_i)$, then for any set $\mathcal{S} \subseteq [n]$ of size s it holds that

$$\text{Dec}(\{y_i\}_{i \in \mathcal{S}}) = F(z; x).$$

- **Computational security:** Security of an HSS HSS is defined via the experiment $\text{Exp}_{\text{HSS},\mathcal{A},I}^{\text{HSS}}$ where the adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ corrupts a set $\mathcal{M} \subset [n]$ of $s-1$ servers. Then, we require

$$\left| \Pr[\text{Exp}_{\text{HSS},\mathcal{A},\mathcal{M}}^{\text{HSS}}(\kappa) = 1] - \frac{1}{2} \right| \leq \text{negl}(\kappa),$$

where the experiment is defined as follows.

Experiment $\text{Exp}_{\text{HSS},\mathcal{A},I}^{\text{HSS}}(\kappa)$ <hr/> $(x_0, x_1) \leftarrow \mathcal{A}_0(1^\kappa)$, where $ x_0 = x_1 $. $b \in_R \{0, 1\}$ $(\hat{x}_1, \dots, \hat{x}_n) \leftarrow \text{Share}(1^\kappa, x_b)$ $b' \leftarrow \mathcal{A}_1(\{\hat{x}_i\}_{i \in I})$ return $b = b'$
--

A trivial construction of the Eval algorithm is the identity function. Then, the Dec algorithm first reconstructs x and computes $F(z; x)$ next. As described above, we utilize an

HSS to perform the Extract algorithm of a VIBE scheme in a distributed way. In this scenario, the Eval algorithm being the identity function means that the party that should learn the identity key also learns the master secret key. Since this is an undesired effect, we impose an additional requirement on the decoding algorithm. We define a *linear decoding* HSS as a slightly weakening of an additive HSS as defined by Boyle et al. [29]. Intuitively, a linear decoding HSS requires the decoding to be a linear combination of the output shares. In contrast to an additive HSS, a linear decoding HSS enables a decoding algorithm whose output depends on the set of servers from which shares are obtained. In particular, the coefficients depend on the servers' indices that computed the shares. This notion allows to capture any s -out-of- n Shamir's secret sharing.

Definition 14 (Linear Decoding HSS). An (s, n) -HSS scheme $\text{HSS} = (\text{Share}, \text{Eval}, \text{Dec})$ is called linear decoding if Dec works as follows:

Let $\{y_1, \dots, y_n\}$ be a set of output shares. Then, for any set $S \subseteq [n]$ of size s , there exists a set of s coefficient $\{a_{S,i}\}_{i \in S}$ such that

$$\text{Dec}(\{y_i\}_{i \in S}) = \sum_{i \in S} a_{S,i} \cdot y_i.$$

E. Threshold IBE

In this section, we state the formal security games for threshold identity-based encryption schemes (TIBE). The notation for TIBE is given in Section II-C. We first define the security game for anonymity and security against chosen-identity attacks.

Experiment $\text{Exp}_{\text{TIBE}, \mathcal{A}}^{\text{A-T-CPA}}(1^\kappa)$

$\mathcal{M} \leftarrow \mathcal{A}_0(1^\kappa)$, where $|\mathcal{M}| < s$
 $\alpha, \beta \leftarrow \{0, 1\}$
 $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]}) \leftarrow \text{Setup}(1^\kappa, s, n)$
 $(\text{ID}_0, \text{ID}_1, m_0, m_1) \leftarrow \mathcal{A}_1^{\mathcal{O}(\cdot, \cdot)}(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in \mathcal{M}})$
 $c^* \leftarrow \text{Encrypt}(\text{pk}, \text{ID}_\alpha, m_\beta)$
 $(\alpha', \beta') \leftarrow \mathcal{A}_2^{\mathcal{O}(\cdot, \cdot)}(c^*)$
return $(\alpha, \beta) = (\alpha', \beta')$

The adversary can use its oracle $\mathcal{O}(\cdot, \cdot)$ to make key generation queries. To do so, the adversary sends (i, id) to \mathcal{O} and receives $(i, \text{ik}_i) \leftarrow \text{ShareKeyGen}(\text{pk}, i, \text{sk}_i, \text{id})$. In the game, we require that ID_0 and ID_1 was not used in any oracle query of \mathcal{A}_1 before or after providing the identities and messages.

Next, we define the game for key generation consistency.

Experiment $\text{Exp}_{\text{TIBE}, \mathcal{A}}^{\text{KC-CPA}}(1^\kappa)$

$\mathcal{M} \leftarrow \mathcal{A}_0(1^\kappa)$, where $|\mathcal{M}| < s$
 $(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in [n]}) \leftarrow \text{Setup}(1^\kappa, s, n)$
 $(\text{ID}, c, \{(i, \text{ik}_i)\}_{i \in \mathcal{S}}, \{(i, \text{ik}'_i)\}_{i \in \mathcal{S}'}) \leftarrow \mathcal{A}_1^{\mathcal{O}(\cdot, \cdot)}(\text{pk}, \text{vk}, \{\text{sk}_i\}_{i \in \mathcal{M}})$,
 where $\mathcal{S}, \mathcal{S}' \subseteq [n] \wedge |\mathcal{S}| = s = |\mathcal{S}'|$
if $\forall i \in \mathcal{S} : \text{ShareVf}(\text{pk}, \text{vk}, \text{ID}, i, \text{ik}_i) = \text{true}$
 $\wedge \forall i \in \mathcal{S}' : \text{ShareVf}(\text{pk}, \text{vk}, \text{ID}, i, \text{ik}'_i) = \text{true}$
 $\wedge \text{ik} = \text{Combine}(\text{pk}, \text{vk}, \text{ID}, \{\text{ik}_i\}_{i \in \mathcal{S}})$
 $\wedge \text{ik}' = \text{Combine}(\text{pk}, \text{vk}, \text{ID}, \{\text{ik}'_i\}_{i \in \mathcal{S}'})$
 $\wedge \text{ik}, \text{ik}' \neq \perp$
 $\wedge \text{Decrypt}(\text{pk}, \text{ID}, \text{ik}, c) \neq \text{Decrypt}(\text{pk}, \text{ID}, \text{ik}', c)$
return 1
else
return 0

The adversary can use its oracle $\mathcal{O}(\cdot, \cdot)$ in the same way as described above without any restrictions on the queried identities.

Definition 15 (ANON-IND-ID-CPA). A TIBE scheme TIBE is ANON-IND-ID-CPA secure if for every $\kappa, n \in \mathbb{N}$, every $1 \leq s \leq n$ and for every PPT adversary $\mathcal{A} := (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$ there exist two negligible function negl_0 and negl_1 such that

$$\left| \Pr[\text{Exp}_{\text{TIBE}, \mathcal{A}}^{\text{A-T-CPA}}(\kappa) = 1] - \frac{1}{4} \right| \leq \text{negl}_0(\kappa) \quad \wedge$$

$$\Pr[\text{Exp}_{\text{TIBE}, \mathcal{A}}^{\text{KC-CPA}}(\kappa) = 1] \leq \text{negl}_1(\kappa).$$

E. POSE: Practical Off-chain Smart Contract Execution

This chapter corresponds to the following publication. The full version is available at [94].

- [95] T. Frassetto, P. Jauernig, D. Koisser, D. Kretzler, B. Schlosser, S. Faust, and A. Sadeghi. “POSE: Practical Off-chain Smart Contract Execution”. In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. 2023. **Part of this thesis.**

POSE: Practical Off-chain Smart Contract Execution

Tommaso Frassetto*, Patrick Jauernig*, David Koisser*, David Kretzler†,
Benjamin Schlosser†, Sebastian Faust† and Ahmad-Reza Sadeghi*

Technical University of Darmstadt, Germany

*first.last@trust.tu-darmstadt.de

†first.last@tu-darmstadt.de

Abstract—Smart contracts enable users to execute payments depending on complex program logic. Ethereum is the most notable example of a blockchain that supports smart contracts leveraged for countless applications including games, auctions and financial products. Unfortunately, the traditional method of running contract code *on-chain* is very expensive, for instance, on the Ethereum platform, fees have dramatically increased, rendering the system unsuitable for complex applications. A prominent solution to address this problem is to execute code *off-chain* and only use the blockchain as a trust anchor. While there has been significant progress in developing off-chain systems over the last years, current off-chain solutions suffer from various drawbacks including costly blockchain interactions, lack of data privacy, huge capital costs from locked collateral, or supporting only a restricted set of applications.

In this paper, we present *POSE*—a practical off-chain protocol for smart contracts that addresses the aforementioned shortcomings of existing solutions. *POSE* leverages a pool of Trusted Execution Environments (TEEs) to execute the computation efficiently and to swiftly recover from accidental or malicious failures. We show that *POSE* provides strong security guarantees even if a large subset of parties is corrupted. We evaluate our proof-of-concept implementation with respect to its efficiency and effectiveness.

I. INTRODUCTION

More than a decade ago, Bitcoin [47] introduced the idea of a decentralized cryptocurrency, marking the advent of the blockchain era. Since then, blockchain technologies have rapidly evolved and a plethora of innovations emerged with the aim to replace centralized platform providers by distributed systems. One particularly important application of blockchains concerns so-called *smart contracts*, complex transactions executing payments that depend on programs deployed to the blockchain. The first and most popular blockchain platform that supported complex smart contracts is Ethereum [58]. However, Ethereum still falls short of the decentralized “world computer” that was envisioned by the community [51]. For example, contracts are replicated among a large group of miners, thereby severely limiting scalability and leading to high costs. As a result, most contracts used in practice in the Ethereum ecosystem are very simple: 80% of popular contracts consist of less than 211 instructions, and almost half of the most active contracts are simple token

managers [49]. More recently proposed computing platforms in permissionless decentralized settings (e.g., [1], [34]) suffer from similar scalability limitations.

In recent years, numerous solutions have been proposed to address these shortcomings of blockchains, one of the most promising being so-called *off-chain execution* systems. These protocols move the majority of transactions off-chain, thereby minimizing the costly interactions with the blockchain. A large body of work has explored various types of off-chain solutions including most prominently state-channels [46], [26], [22], Plasma [52], [37] and Rollups [48], [5], which are actively investigated by the Ethereum research community. Other schemes use execution agents that need to agree with each other [60], [59], rely on incentive mechanisms [36], [57], or leverage Trusted Execution Environments (TEEs) [20], [25]. A core challenge that arises while designing off-chain execution protocols is to handle the possibility of parties who stop responding, either maliciously or accidentally. Without countermeasures, this may cause the contract execution to stop unexpectedly, which violates the *liveness* property. Despite major progress towards achieving liveness in a off-chain setting, current solutions come with at least one of these limitations: **[i]** participating parties need to lock large amounts of collateral; **[ii]** costly blockchain interactions are required at every step of the process or at regular intervals; and finally **[iii]** the set of participants and the lifetime need to be known beforehand, which limits the set of applications supported by the system. Additionally, existing solutions often **[iv]** do not support keeping the contract state confidential, which is required, e.g., for eBay-style proxy auctions [9] and games such as poker. We refer the reader to Table II for an overview on related work and to Section X for a detailed discussion.

Addressing all of these limitations in one solution while guaranteeing liveness is highly challenging. Currently, there are two ways to address the risks of unresponsive parties. The first approach is to require collateral, i.e., parties have to block large amounts of money, which is used to disincentivize malicious behavior and to compensate parties in case of premature termination (cf. **[i]**). Since the amount of collateral depends on the number of participants and the amount of money in the contract, both must be fixed for the whole lifetime of the contract. To ensure payout of the collateral, the lifetime of the contract must be fixed as well (cf. **[iii]**). The second approach is to store contract state on the blockchain to enable other parties to resume execution. However, this is both expensive and leads to long waiting times due to frequent synchronization with the blockchain (cf. **[ii]**). Further, if the contract state needs to be confidential, and hence, is not publicly verifiable,

verifying the correctness of the contract execution is harder (cf. [iv]). Realizing a system tackling all these challenges in a holistic way could pave the way towards the envisioned “world computer”. We will further elaborate on the specific challenges in Section III.

Our goals and contributions: We present *POSE*, a novel off-chain execution framework for smart contracts in permissionless blockchains that overcomes these challenges, while achieving correctness and strong liveness guarantees. In *POSE*, each smart contract runs on its own subset of TEEs randomly selected from all TEEs registered to the network. One of the selected TEEs is responsible for the execution of a smart contract.

However, as the system hosting the executing TEE may be malicious (e.g., the TEE could simply be powered off during contract execution), our protocol faces the challenge of dealing with malicious operator tampering, withholding and replaying messages to/from the TEE. Hence, the TEE sends state updates to the other selected TEEs, such that they can replace the executing TEE if required. This makes *POSE* the first off-chain execution protocol with strong liveness guarantees. In particular, liveness is guaranteed as long as at least one TEE in the execution pool is responsive. Due to this liveness guarantee, there is no inherent need for a large collateral in *POSE* (cf. [i]). The state remains confidential, which allows *POSE* to have private state (cf. [iv]). Furthermore, *POSE* allows participants to change their stake in the contract at any time. Thus, *POSE* supports contracts without an a-priori fixed lifetime and enables the set of participants to be dynamic (cf. [iii]). Above all, *POSE* executes smart contracts quickly and efficiently without any blockchain interactions in the optimistic case (cf. [ii]).

This enables the execution of highly complex smart contracts and supports emerging applications to be run on the blockchain, such as federated machine learning. Thus, *POSE* improves the state of the art significantly in terms of security guarantees and smart contract features. To summarize, we list our main contributions below:

- We introduce *POSE*, a fast and efficient off-chain smart contract execution protocol. It provides strong guarantees without relying on blockchain interactions during optimistic execution, and does not require large collaterals. Moreover, it supports contracts with an arbitrary contract lifetime and a dynamic set of users. An additional unique feature of *POSE* is that it allows for confidential state execution.
- We provide a security analysis in a strong adversarial model. We consider an adversary which may deviate arbitrarily from the protocol description. We show that *POSE* achieves correctness and state privacy as well as strong liveness guarantees under static corruption, even in a network with a large share of corrupted parties.
- To illustrate the feasibility of our scheme, we implement a prototype of *POSE* using ARM TrustZone as the TEE and evaluated it on practical smart contracts, including one that can merge models for federated machine learning in 238ms per aggregation.

II. ADVERSARY MODEL

The goal of *POSE* is to allow a set of users to run a complex smart contract on a number of TEE-enabled systems. Note, that *POSE* is TEE-agnostic and can be instantiated on any TEE architecture adhering to our assumptions, similar to, e.g., FastKitten [25]. In order to model the behavior and the capabilities of every participant of the system, we make the following assumptions:

A1: We assume the TEE to protect the enclave program, in line with other TEE-assisted blockchain proposals [63], [25], [20], [17], [64], [43]. Specifically:

A1.1: We assume the TEE to provide integrity and confidentiality guarantees. This means that the TEE ensures that the enclave program runs correctly, is not leaking any data, and is not tampering with other enclaves. While our proof of concept is based on TrustZone, our design does not depend on any specific TEE. In practice, the security of a TEE is not always flawless, especially regarding information leaks. However, plenty of mitigations exist for the respective commercial TEEs; hence, we consider the problem of information leakage from any specific TEE, as well as TEE-specific vulnerabilities in security services, orthogonal to the scope of this paper. We discuss some mitigations to side-channel attacks to TrustZone, as well as the possible grave consequences of a compromised or leaking TEE for the executed smart contract, in Section VII-B.

A1.2: We further assume the adversaries to be unable to exploit memory corruption vulnerabilities in the enclave program. This could be ensured using a number of different approaches, e.g., by using memory-safe languages, by deploying a run-time defense like CFI [11], or by proving the correctness of the enclave program using formal methods. The existence of these defenses can be proven through remote attestation (cf. A3).

A2: We assume the TEE to provide a good source of randomness to all its enclaves and to have access to a relative clock according to the GlobalPlatform TEE specification [32].

A3: We assume the TEE to support *secure remote attestation*, i.e., to be able to provide unforgeable cryptographic proof that a specific program is running inside of a genuine, authentic enclave. Further, we assume the attestation primitive to allow differentiation of two enclaves running the same code under the same data. Note that today’s industrial TEEs support remote attestation [3], [6], [8], [35], [56].

A4: We assume the TEE operators, i.e., the persons or organizations owning the TEE-enabled machines, to have full control over those machines, including root access and control over the network. The operators can, for instance, provide wrong data to an enclave, delay the transmission of messages to it, or drop messages completely. The operators can also completely disconnect an enclave from the network or (equivalently) power off the machine containing it. However, as stated in A1.1, the operators cannot leak data from any enclave or influence its computation in any way besides by sending (potentially malicious) messages to it through the official software interfaces.

A5: We assume static corruption by the adversary. More precisely, a fixed fraction of all operators is corrupted while an arbitrary number of users can be malicious (including the case where they all are). We model each of the malicious parties as *byzantine adversaries*, i.e., they can behave in arbitrary ways.

A6: We assume the blockchain used by the parties to satisfy

the following standard security properties: common prefix (ignoring the last γ blocks, honest miners have an identical chain prefix), chain quality (blockchain of honest miner contains significant fraction of blocks created by honest miners), and chain growth (new blocks are added continuously). These properties imply that valid transactions are included in one of the next α blocks and that no valid blockchain fork of length at least γ can grow with the same block creation rate as the main chain. We deem protection against network attacks (e.g., network partition attacks), which violate these standard properties, orthogonal to our work.

III. DESIGN

POSE is a novel off-chain protocol for highly efficient smart contract execution, while providing strong correctness, privacy, and liveness guarantees. To achieve this, *POSE* leverages the integrity and confidentiality guarantees of TEEs to speed up contract execution and make significantly more complex contracts practical¹. This is in contrast to executing contracts on-chain, where computation and verification is distributed over many parties during the mining process. *POSE* supports contracts with arbitrary lifetime and number of users, which includes complex applications like the well-known CryptoKitties [2]. We elaborate more on interaction between contracts in Appendix B. Our protocol involves users, operators and a single on-chain smart contract. *Users* aim to interact with smart contracts by providing inputs and obtaining outputs in return. *Operators* own and manage the TEE-enabled systems and contribute computing power to the *POSE* network by creating protected execution units, called *enclaves*, using their TEEs. These enclaves perform the actual state transitions triggered by users. A simple on-chain smart contract, which we call *manager*, is used to manage the off-chain enclave execution units. In the optimistic case, when all parties behave honestly, *POSE* requires only on-chain transactions for the creation of a *POSE* contract as well as the locking and unlocking of user funds. The smart contract execution itself is done without any on-chain transactions.

A. Architecture Overview

Figure 1 illustrates the high-level working of *POSE*. Before contract creation, there is already a set of enclaves that are registered with the on-chain manager contract. The registration process is explained in detail in Section V-E1. To create a *POSE* contract, a user will initialize a contract creation with the manager (Step 1), which includes a chosen enclave—out of the registered set—to execute the off-chain contract creation. In Step 2, the chosen *creator* enclave will setup the *execution pool* for the given smart contract. In Figure 1, the pool size is set to three; thus, the *creator* enclave will randomly select three enclaves from the set of all enclaves registered in the system (Step 3). In Step 4, the *creator* enclave will submit the finalized contract information to the manager. This includes the composition of the execution pool, i.e., a selected *executor* enclave, which is responsible for executing the *POSE* contract, as well as the *watchdogs*, ensuring availability. We elaborate on this in-depth in Section V-E2. In Step 5, another user can

¹We design *POSE* without depending on any specific TEE implementation. In Section VII-B, we discuss the implications of using ARM TrustZone to realize our scheme.

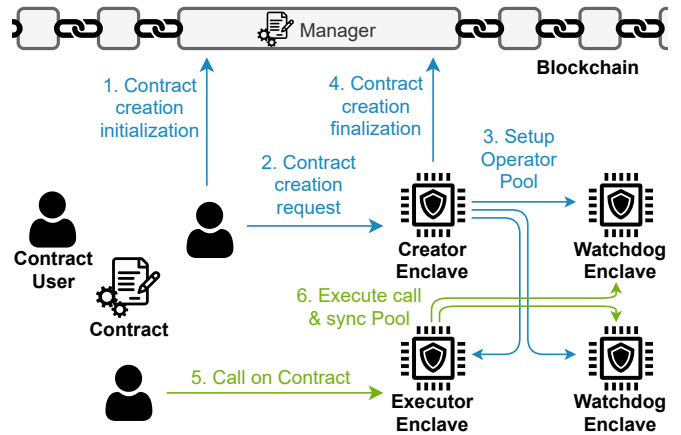


Fig. 1. Exemplary overview how *POSE* contracts are created (in blue) and executed (in green).

now call the new contract by directly contacting the executor. Finally, for Step 6, the executor will execute the user’s contract call and distribute the resulting state to the watchdog enclaves, which confirm the state update. See Section V-E3 for a detailed specification of the execution protocol. If one of the enclaves stops participating (e.g., due to a crash), the dependent parties can challenge the enclave on the blockchain (see Section V-E4). The dependent party can either be the user awaiting response from the executor or the executor waiting for the watchdogs’ confirmation. For example, if the executor stops executing the contract, the executor is challenged by the user. A timely response constitutes a successful state transition as requested by the user. Otherwise, if the current executor does not respond, one of the watchdogs will fill in as the new executor. This makes *POSE* highly available, as long as at least one watchdog enclave is dependable; thus, avoiding the need for collateral to incentivize correct behavior. Further, *POSE* supports private state, as the state is only securely shared with other enclaves.

B. Design Challenges

We encountered a number of challenges while designing *POSE*. We briefly discuss them below.

Protection Against Malicious Operators. *POSE*’s creator, executor, and watchdogs are protected in isolated enclaves running within the system, which is itself still under control of a potentially malicious operator. Hence, operators can provide arbitrary inputs, modify honest users’ messages, execute replay attacks, and withhold incoming messages. Moreover, the system and its TEE (i.e., enclaves) can be turned off completely by its operator. In order to protect honest users from malicious operators, we incorporate several security mechanisms. While malicious inputs and modification of honest users’ messages can easily be prevented using standard measures like a secure signature scheme, preventing withholding of messages is more challenging. One particular reason is that for unreceived messages, an enclave cannot differentiate between unsent and stalled messages by the operator. Hence, we incorporate an on-chain challenge-response procedure, which provides evidence about the execution request and the existence of a response to the enclave.

Achieving Strong Liveness Guarantees. We enable dependent parties to challenge unresponsive operators via the blockchain. The challenged operators either provide valid responses over the blockchain that dependent parties can use to finalize the state transition, or they are dropped from the execution pool. In case an executor operator has been dropped, we use the execution pool to resume the execution; this requires state updates to be distributed to all watchdogs. With at least one honest operator in the execution pool, the pool will produce a valid state transition. Our protocol tolerates a fixed fraction of malicious operators as stated in our adversary model (cf. Section II). By selecting the pool members randomly, we guarantee with high probability that at least one enclave—controlled by an honest operator—is part of the execution pool. We show in Section VII-A that our protocol achieves strong liveness guarantees.

Synchronization with the Blockchain. Some of the actions taken by an enclave depend on blockchain data, e.g., deposits made by clients. Hence, it is crucial to ensure that the blockchain data available to an enclave is consistent and synchronized with the main chain. As an enclave does not necessarily have direct access to the (blockchain) network, it has to rely on the blockchain data provided by the operator. However, the operator can tamper with the blockchain data and, e.g., withhold blocks for a certain time. Thus, a major challenge is designing a synchronization mechanism that (i) imposes an upper bound on the time an enclave may lag behind the main chain, (ii) prevents an operator from isolating an enclave onto a fake side-chain, and (iii) ensures correctness and completeness of the blockchain data provided to the enclave, without (iv) requiring the enclave to validate or store the entire blockchain. We present our synchronization mechanism addressing these challenges in Section V-D.

Reducing Blockchain Interactions. Our system aims to minimize the necessary blockchain interactions to avoid expensive on-chain computations. In the optimistic scenario, the only on-chain transactions necessary are the contract creation and the transfer of coins. The transfer transactions can also be bundled to further reduce blockchain interactions. Note that the virtualization paradigm known from state channels [26] can be applied to our system. This enables parties to install virtual smart contracts within existing smart contracts, and hence, without any on-chain interactions at all. In the pessimistic scenario, i.e., if operators fail to provide valid responses, they have to be challenged, which requires additional blockchain interactions.

Support of Private State. To support private state of randomized contracts, careful design is required to avoid leakage. While the confidentiality guarantees of TEEs prevent any data leakage during contract execution, our protocol needs to ensure that an adversary cannot learn any information except the output of a successful execution. In particular, in a system where the contract state is distributed between several parties, we need to prevent the adversary from performing an execution on one enclave, learning the result, and exploiting this knowledge when rolling back to an old state with another enclave. This is due to the fact that a re-execution may use different randomness or different inputs resulting in a different output. We prevent these attacks by outputting state updates to the users only if all pool members are aware of the new

state. Moreover, by solving the challenge of synchronization between enclaves and the blockchain, we prevent an adversary from providing a fake chain to the enclave, in which honest operators are kicked from the execution pool. Such a fake chain would allow an attacker to perform a parallel execution. While results of the parallel (fake) execution cannot affect the real execution, they can prematurely leak private data, e.g. the winner in a private auction.

IV. DEFINITIONS & NOTATIONS

In the following, we introduce the cryptography primitives, definition, and notations used in the *POSE* protocol.

Cryptographic Primitives. Our protocol utilizes a public key encryption scheme $(GenPK, Enc, Dec)$, a signature scheme $(GenSig, Sign, Verify)$, and a secure hash function $H(\cdot)$. All messages sent within our protocol are signed by the sending party. We denote a message m signed by party P as $(m; P)$. The verification algorithm $Verify(m')$ takes as input a signed message $m' := (m; P)$ and outputs ok if the signature of P on m is valid and bad otherwise. We identify parties by their public keys and abuse notation by using P and P 's public key pk_P interchangeably. This can be seen as a direct mapping from the identity of a party to the corresponding public key.

TEE. We comprise the hardware and software components required to create confidential and integrity-protected execution environments under the term TEE. An operator can instruct her TEE to create new *enclaves*, i.e., new execution environments running a specified program. We follow the approach of Pass et al. [50] to model the TEE functionality. We briefly describe the operations provided by the ideal functionality formally specified in [50, Fig. 1]. A TEE provides a $TEE.install(prog)$ operation which creates a new enclave running the program $prog$. The operation returns an enclave id eid . An enclave with id eid can be executed multiple times using the $TEE.resume(eid, inp)$ operation. It executes $prog$ of eid on input inp and updates the internal state. This means in particular that the state is stored across invocations. The $resume$ operation returns the output out of the program. We slightly deviate from Pass et al. [50] and include an attestation mechanism provided by a TEE that generates an attestation quote ρ over $(eid, prog)$. ρ can be verified by using method $VerifyQuote(\rho)$. We consider only one instance \mathcal{E} running the *POSE* program per TEE. Therefore, we simplify the notation and write $\mathcal{E}(inp)$ for $TEE.resume(eid, inp)$.

Blockchain. We denote the blockchain by BC and the average block time by τ . A block is considered final if it has at least γ confirmation blocks. Throughout the protocol description in Section V-E, enclaves consider only transactions included in final blocks. Finally, we define that any smart contract deployed to the blockchain is able to access the current timestamp using the method $BC.now$ and the hash of the most recent 265 blocks [7] using the method $BC.bh(i)$ where i is the number of the accessed block. These features are available on Ethereum.

V. THE *POSE* PROTOCOL

The *POSE* protocol considers four different roles: a manager smart contract deployed to the blockchain, operators that run TEEs, enclaves that are installed within TEEs, and users

that create and interact with *POSE* contracts. In the following, we will shortly elaborate on the on-chain smart contract and the program executed by the enclaves, explain the *POSE* protocol, and finally explain further security mechanisms that are omitted in the protocol description.

A. Manager

We utilize an on-chain smart contract in order to manage the *POSE* system’s on-chain interactions. We call this smart contract *manager* and denote it by M . On the one hand, M keeps track of all registered *POSE* enclaves. This enables the setup of an execution pool whenever an off-chain smart contract instance is created. On the other hand, it serves as a registry of all *POSE* contract instances. M stores parameters about each contract to determine the instance’s status. We denote the tuple describing a contract with identifier id as M^{id} . In particular, the manager stores the creator enclave (*creator*), a hash of the program code (*codeHash*), the set of enclaves forming the execution pool (*pool*), a total amount of locked coins (*balance*), and a counter of withdrawals (*payouts*). We set the field *creator* to \perp after the creation process has been completed to identify that a contract is ready to be executed. Moreover, for both executor and watchdog challenges, the contract allocates storage for a tuple containing the challenge message (*c1Msg* resp. *c2Msg*), responses (*c1Res* resp. *c2Res*), and the timestamp of the challenge submission (*c1Time* resp. *c2Time*). A non-empty field *c1Time* resp. *c2Time* signals that there is a running challenge.

Every *POSE* enclave maintains a local version of the manager state extracted from the blockchain data it receives from the operator when being executed. This enables all enclaves to be aware of on-chain events, e.g., ongoing challenges.

B. POSE Program

All enclaves registered within the system run the *POSE* program that enforces correct execution and creation of *POSE* contracts. In practice, the *POSE* program’s source code will be publicly available, e.g., in a public repository, so that the community can audit it. Our protocol ensures that all registered enclaves run this code using remote attestation (cf. Section V-E1: Enclave registration). We present methods required for the execution protocol in Program 1 and defer methods for the contract creation to the full version of this paper [31].

Whenever an enclave is invoked, it synchronizes itself with the blockchain network and receives the relevant blockchain data in a reliable way (cf. Section V-D). This way, the *POSE* program has access to the current state of the manager. In order to support arbitrary contracts, we define a common interface in Section V-C that is used by the *POSE* program to invoke contracts.

Enclaves running the *POSE* program only accept signed messages as input. The public keys of pool members for signature verification are derived from the synchronized blockchain data. According to our adversary model (cf. Section II), the adversary cannot read or tamper messages originating from honest users or the enclave itself. Further, the contracts themselves keep track of already received execution requests and do not perform state transitions for duplicated requests.

Program 1: POSE Program (execution) executed by enclave T

```

Upon invocation with input blockchain data BC, store BC.
Upon receiving  $m := (\text{execute}, id, r, move; U)$ , do:
  1) If  $M^{id}.pool[0] \neq T$  or  $\mathcal{T}_{wait}^{id} \neq \emptyset$ , return (bad).
  2) Execute  $C_{id}.nextState(U, BC, move, H(m))$ .
  3) Store  $\mathcal{T}_{wait}^{id} = M^{id}.pool$  and  $h^{id} = H(m)$ , set
      $c = Enc(C_{id}.getState(all); key^{id})$  and return
      $(\text{update}, id, c, h^{id}; T)$ .
Upon receiving  $m := (\text{update}, id, c, h; T')$ , do:
  1) If  $T' \neq M^{id}.pool[0]$  or  $T \notin M^{id}.pool$ , return (bad).
  2) Define  $state = Dec(c; key^{id})$  and call
      $C_{id}.update(state, h)$ .
  3) Return  $(\text{confirm}, id, h; T)$ .
Upon receiving  $\{m_i := (\text{confirm}, id, h_i; T_i)\}_i$ , do:
  1) If  $M^{id}.pool[0] \neq T$  or  $\mathcal{T}_{wait}^{id} = \emptyset$ , return (bad).
  2) Set  $\mathcal{T}_{wait}^{id} = \mathcal{T}_{wait}^{id} \cap M^{id}.pool$ .
  3) For each  $m_i$  do:
     • If  $h_i \neq h^{id}$  or  $T_i \notin \mathcal{T}_{wait}^{id}$ , skip  $m_i$ .
     • Otherwise remove  $T_i$  from  $\mathcal{T}_{wait}^{id}$ .
  4) If  $\mathcal{T}_{wait}^{id} \neq \{T\}$ , return (bad). Otherwise, set  $\mathcal{T}_{wait}^{id} = \emptyset$ ,
      $state := C_{id}.getState(pub)$  and return
      $(\text{ok}, id, state, h^{id}; T)$ .

```

(cf. Section V-C). This prevents replay attacks against both, executive and watchdog enclaves.

C. POSE Contracts

Although our system supports the execution of arbitrary smart contracts, the contracts need to implement a specific interface (cf. Program 2). This allows any *POSE* enclave to trigger the execution without knowing details about the smart contract functionality. Upon an execution request from some user, the *POSE* enclave provides the user’s identity U , blockchain data BC, the description of the user’s request, *move*, and the request hash, h , to the smart contract’s method *nextState*. The smart contract first processes the relevant blockchain data and marks the current length of the blockchain as processed. This feature is mainly used to enable smart contracts to deal with money, i.e., to detect on-chain deposits and withdrawals. We elaborate on the processing of blockchain data in Section V-D, and on the money mechanism of the *POSE* system in Appendix E. Note that double spending within a contract is prevented due to sequential processing of any execution request, and double spending of on-chain payouts is prevented by the mechanism explained in Appendix E. After the blockchain data is processed, *nextState* executes the move requested by the user and updates the state accordingly. Method *update* takes state *new* and hash h (for preventing replay attacks) as input and sets *new* as the contract state. This includes the length of the blockchain that is marked as processed. Further, the smart contract provides method *getState*. If called with *flag* = *all*, it returns the whole smart contract state. Otherwise, if called with *flag* = *pub*, it returns only the public state. In order to prevent replay attacks, each smart contract maintains a list with the hashes of already received execution requests, *Rec*. In case of duplicated requests, i.e., $h \in Rec$, both the *nextState* method and the *update* method, do not perform any state transition. Instead, they interpret the request as a dummy move that has no effect on the state. If executed successfully, the *nextState* method

Program 2: Interface of a contract C executed within a POSE enclave

Function: $nextState(U, BC, move, h)$
 Function: $update(new, h)$
 Function: $getState(flag)$

adds the executed request to Rec , i.e., $Rec = Rec \cup \{h\}$. As Rec is part of the state, it is updated by the $update$ method as well. While it might seem counter intuitive to overwrite the list of received requests, this feature is required to ensure that all enclaves are aware of the same transition history; even if an executor distributes a state update to just a subset of watchdogs before getting kicked ².

We consider the initial state of a smart contract to be hard-coded into the smart contract description. If an enclave creates a new smart contract instance, the initial state is automatically initialized. A contract state additionally contains a variable to store the highest block number of the already processed blockchain data. This variable is used to detect which transactions of received blockchain data have already been handled.

D. Synchronization

As some of the actions taken by an enclave depend on blockchain data, e.g., deposits to the contract, it is crucial to ensure that the blockchain state available to a registered enclave \mathcal{E} is consistent and synchronized with the main chain. In particular, blocks that are considered final by some party, will eventually be considered final by all parties. We design a synchronization mechanism that allows \mathcal{E} to synchronize itself without having to validate whole blocks. Note that \mathcal{E} has access to a relative time source according to our adversary model (see Section II).

Upon initialization, \mathcal{E} receives a chain of block headers BCH of length $\gamma + 1$. Note that the first block p of BCH can be considered final since it has γ confirmation blocks. First, \mathcal{E} checks that BCH is consistent in itself and sets its own clock to be the one of the latest block's timestamp. Second, \mathcal{E} signs block p as blockchain evidence that needs to be provided to the manager. The registration mechanism (cf. Section V-E1) uses this evidence to ensure that \mathcal{E} has been initialized with a valid sub-chain of the main-chain up to block p . Further, the registration mechanism checks that p is at most τ_{slack}^{on} blocks behind the current one; τ_{slack}^{on} needs to account for the confirmation blocks and the fact that transactions are not always mined immediately. Via this parameter, we can set an upper bound to the time τ_{slack}^{off} an enclave may lag behind; τ_{slack}^{off} additionally considers potential block variance and the fact that miners have some margin to set timestamps. In the following, we call τ_{slack}^{off} *slack* ³. Clients that want a contract execution to capture on-chain effects, e.g., deposits, wait until

the enclave considers the corresponding block as final, even when being at slack.

Once successfully initialized, \mathcal{E} synchronizes itself with the blockchain. Whenever a registered enclave is executed throughout the protocol, it receives the sub-chain of block headers BCH' that have been mined since the last execution. \mathcal{E} checks that BCH' is a valid successor of BCH where blocks in BCH that have not been final may change. Further, \mathcal{E} checks that the latest block in BCH' is at most $\tau_{variance}$ behind the own clock; $\tau_{variance}$ captures the variance in the block creation time and the fact that miners have some margin to set timestamps. When receiving a block that is before the own clock, the clock is adjusted.

Finally, we need to prevent an operator from isolating its enclave by setting up a valid sidechain with manipulated timestamps. To this end, we require the operators to periodically provide new blocks to \mathcal{E} even if \mathcal{E} does not need to take any action. In particular, we require that the operator provides at least L blocks within time τ_p where τ_p accounts for potential block time variances. The system is secure as long as the attacker cannot mine L blocks within time τ_p while the honest miners can. Hence, the selection of τ_p and L has some implications on the fraction of adversarial computing power that can be tolerated by the system. Since 2018, an interval of 50 (100, 200, 300) blocks took at most 33 (28, 26, 25) seconds per block [10], which might all be reasonable choices for L and $\frac{\tau_p}{L}$. As the average block time is around 13 seconds [4], the adversary gets 2 – 3 times more time to mine the blocks of its sidechain. This means that the system can tolerate adversarial fractions from a third (when instantiated with $L = 300$ and $\tau_p = 25 \cdot L$) to a forth (when instantiated with 50 and $33 \cdot L$).

While the above techniques allow an enclave to synchronize itself, the enclave does not have access to the block data, yet. Instead of requiring enclaves to validate whole blocks, we require operators to filter the relevant transactions and provide them to the enclave while enabling the enclaves to check correctness and completeness of the received data itself. For the latter, we introduce *incrTxHash*, a hash maintained by the manager and all initialized enclaves that is based on all relevant transactions. Whenever the manager receives a relevant transaction tx , it updates *incrTxHash*, such that *incrTxHash* _{$i+1$} is defined as

$$H(\text{incrTxHash}_i \parallel tx.data \parallel tx.sender \parallel tx.value)$$

where $tx.data$ is the raw data of tx , $tx.sender$ denotes the creator of tx , and $tx.value$ contains the amount of any deposits or withdrawals. Whenever enclaves are invoked with new blocks, operators additionally provide all relevant transactions. This way, enclaves can re-compute the new incremental hash and compare the result to the on-chain value of *incrTxHash*. In order to verify that the on-chain *incrTxHash* is indeed part of the main chain, operators additionally provide a Merkle proof showing that *incrTxHash* is part of the state tree. The proof can be validated using the state root, which is part of the block headers provided to the enclaves. This way, enclaves can ensure that operators have not omitted or manipulated any relevant transactions.

²In practice, the state update removes at most the last element from the request history; a fact that can be exploited to reduce the size of state updates.

³We can reduce the slack assuming an absolute source of time realized via trusted NTP servers, cf. [20], by enabling the enclave to check if she was invoked with the most recent block headers up to some variance of the timestamps.

E. Protocol Description

In this section, we dive into a detailed description of our protocol. We present 1) enclave registration, 2) contract creation, 3) contract execution, and 4) the challenge-response parts of our protocol. The *POSE* program running inside the operators' enclaves is stated in Section V-B. For the sake of exposition, we extracted the validation steps performed by the manager on incoming messages into Program 3 in Appendix C. Further, we elaborate in Appendix E on the coin flow within the protocol.

1) *Enclave Registration*: Operator O controlling some TEE unit can contribute to the *POSE* system by instructing his TEE to create a new *POSE* enclave \mathcal{E}_O . The protected execution environment \mathcal{E}_O needs to be initialized with the *POSE* program presented in Section V-B. During the creation of \mathcal{E}_O , an asymmetric key pair (pk_O, sk_O) is generated. The secret key sk_O is stored inside the enclave and hence is only accessible by the *POSE* program running in \mathcal{E}_O . The public key pk_O is returned as output to the operator. Furthermore, operator O uses the TEE to produce an attestation ρ_O stating that the freshly generated enclave \mathcal{E}_O runs the *POSE* program and controls the secret key corresponding to pk_O .⁴

Finally, O sends the latest $\gamma+1$ block headers BCH together with the relevant blockchain data to the enclave which validates the consistency of the block headers and completeness of the blockchain data (cf. Section V-D) and returns a blockchain evidence ρ_O^{BC} , i.e., a signed tuple containing the blockhash and the number of the latest final block known to the enclave. After operator O created a new *POSE* enclave \mathcal{E}_O , O can register \mathcal{E}_O by sending $m := (\text{register}, \mathcal{E}_O, \rho_O, \rho_O^{\text{BC}}; O)$ to manager M . M verifies that ρ_O is a valid attestation and that ρ_O^{BC} refers to a block on the blockchain known to M that is not older than $\tau_{\text{slack}}^{\text{on}}$ blocks. If the check holds and the signature of the operator is valid, i.e., $\text{Verify}(m) = \text{ok}$, M adds \mathcal{E}_O (identified by its public key pk_O) to the set of registered enclaves, i.e., $M.\text{registered} := M.\text{registered} \cup \{\mathcal{E}_O\}$. This procedure ensures that all registered enclaves run the *POSE* program and that the secret key sk_O remains private. Hence, re-attesting enclaves during later protocol steps is not needed.

2) *Contract Creation*: The creation protocol is initiated by a user U who wants to install a new smart contract, with program code $code$, into the *POSE* system. We outline the protocol in the following and provide a full explanation and specification in the full version of this paper [31].

U picks an arbitrary registered enclave \mathcal{E}_C and sends a creation initialization to M containing $H(code)$ and \mathcal{E}_C . The manager M allocates a new contract tuple with a fresh identifier id . Next, U sends a creation request, containing $code$, to \mathcal{E}_C which randomly selects n enclaves for the contract execution pool and samples a symmetric pool key. The generated information is distributed in a confidential way to all pool enclaves, which install a new smart contract with code $code$ and confirm the installation to \mathcal{E}_C . Finally, \mathcal{E}_C signs a creation

⁴An attestation mechanism can be designed based on a chain of trust, where the TEEs manufacturer's public key represents the root. This way a smart contract knowing a list of public keys can verify an attestation quote without further interaction. We omit further details about the practical implementation and refer the reader to [50].

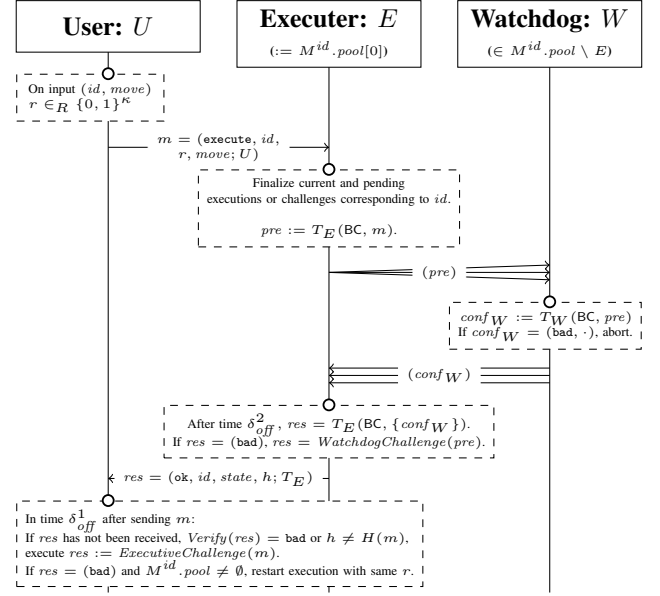


Fig. 2. Detailed execution protocol.

confirmation, which is submitted to M that marks the contract as created.

If the contract is not created within a certain time, U starts a creation challenge. If any pool member does not respond to \mathcal{E}_C timely, \mathcal{E}_C starts a pool challenge (cf. Section V-E4).

3) *Contract Execution*: The execution protocol is initiated by a user U who wants to execute an existing smart contract, identified by id , with input $move$. The protocol is specified in Figure 2. Program 1 specifies the parts of the *POSE* program that are relevant for the contract execution.

To trigger the execution, U sends an execution request to operator E controlling the executor enclave \mathcal{E}_E , the first enclave in the contract pool stored at M . \mathcal{E}_E executes the request and securely propagates the new state to all other pool members, called watchdogs. If any watchdog does not confirm in time, it is challenged by E (cf. *Challenge-Response*). Eventually, \mathcal{E}_E receives confirmations from all watchdogs or the unresponsive watchdogs are kicked out of the pool. Either way, \mathcal{E}_E outputs the new public state to U . We want to stress that this way no party gets to know the result of an update before all pool members agree on the update. If E does not respond in time, it is challenged by U (cf. *Challenge-Response*). If E does not respond to the challenge, it is kicked from the pool by U . The next enclave in the pool, \mathcal{E}'_E , takes over as the new executor. At this point, the new executor might be on a different state than the other pool members, since \mathcal{E}'_E might have received the previous state update but some other pool members not, or vice versa.

Our system automatically ensures that all enclaves share the same contract state after the next successful execution, in which \mathcal{E}'_E distributes its state to the other enclaves. Let us call the previous incompletely distributed update $update$ and the new updated initiated by \mathcal{E}'_E $update'$. In case \mathcal{E}'_E has received $update$, $update'$ is a successor of $update$, and hence, covers both updates. This way, a watchdog that updates to $update'$

essentially contains both executions, $update$ and $update'$. In case \mathcal{E}'_E has not received $update$ but the other watchdogs have, \mathcal{E}'_E either propagates the update already known to the watchdogs, i.e., $update = update'$, or a concurrent one, i.e., $update \neq update'$. For the former, the watchdogs interpret the update as a dummy update without any effect as the corresponding execution request is already within their list of received request hashes (cf. Section V-C). For the latter, the update of the watchdogs is overwritten by the one of the executive enclave. As $update$ has been incomplete, and hence, produced no public output, it is safe to overwrite this update. To produce a public output for $update$, all pool enclaves including \mathcal{E}'_E would have to confirm $update$.

Finally, U can just submit the previous execution request with the same random nonce r to \mathcal{E}'_E . In case the enclave has already seen this request, it is interpreted as empty dummy move which prevents a duplicated execution.

4) *Challenge-Response*: If any party does not receive a *timely* response to its messages during the off-chain execution, it challenges the receiver on-chain. Therefore, all operators need to monitor the blockchain for any on-chain challenges. We will elaborate on the timeouts (δ_{\star}^{\dagger}), where $\dagger \in \{0, 1\}$ and $\star \in \{off, on\}$, which define the notion of *timely* in Appendix D. In particular, we describe the relation between δ_{\star}^1 and δ_{\star}^2 . The challenge-response procedure is executed in all of the following cases.

- The creator enclave has not responded to the user within time δ_{off}^1 during the contract creation protocol.
- At least one pool enclave has not responded to the creator enclave within time δ_{off}^2 during the contract creation protocol.
- The executor enclave has not responded to the user within time δ_{off}^1 during the contract execution protocol.
- At least one watchdog enclave has not responded to the executor enclave within time δ_{off}^2 during the contract execution protocol.

Since (a) is conceptually identically to (c) and (b) to (d), we present the executor challenge (c) and the watchdog challenge (d) in Figure 3 and Figure 4. The specifications of (a) and (b) are provided in the full version of this paper [31].

For the executor challenge as shown in Figure 3, suppose user U has not received a result from the executor enclave \mathcal{E}_E within time δ_{off}^1 , then, U starts the challenge-response protocol. To this end, U sends the execution request to the manager M who verifies the validity of the message (cf. Program 3). If all checks hold, M stores the challenge message and then starts timeout δ_{on}^1 by storing the current timestamp. As soon as the challenge message is recorded on-chain, the operator of the executor enclave \mathcal{E}_E extracts the execution request from the challenge and starts the execution. Performing the execution request is identical to the standard execution as described in Section V-E3. However, the operator prioritizes challenges over off-chain execution requests to avoid getting kicked. Additionally, if \mathcal{E}_E already performed the state update and state propagation, the operator may use the already obtained result as response. Either way, if the operator sends a response message in time, the manager M checks the validity of the message and whether or not it matches the stored challenge. If all checks succeed, M stores the result and removes the

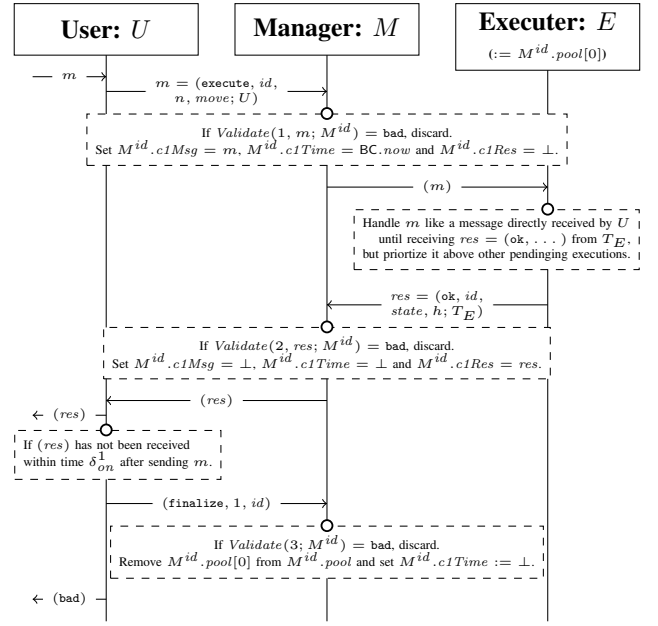


Fig. 3. Detailed executor challenge protocol.

challenge message. This finalizes the challenge procedure. If the operator does not send a valid response in time δ_{on}^1 , user U sends message `finalize` to M . This triggers the manager to kick \mathcal{E}_E from the execution pool of this contract and assign the next enclave in the list as the new executor enclave, if possible. Then, if the pool is not empty, U restarts the execution. As M only accepts a response if the operator executed the challenged request correctly, the described procedure ensures that there is either a consistent state transition or \mathcal{E}_E is kicked from the execution pool, hence, ensuring liveness as long as there remains one active operator.

Since the executor enclave \mathcal{E}_E is dependent on the confirmation message from all watchdog enclaves, it is necessary to allow \mathcal{E}_E to challenge the watchdog enclaves as well (Figure 4). In this case, the executor enclave acts as the challenger and all watchdog enclaves need to provide a confirmation message as response. At the end of this challenge-response protocol, all unresponsive watchdog enclaves are removed from the execution pool. The executor enclave then continues performing the execution with all confirmations obtained during this procedure. Again, M only accepts responses if the watchdog executed the state update correctly, hence, ensuring that a watchdog either performs the correct state update or is kicked from the pool.

F. Security Remarks

To keep the protocol description compact, we omitted some security features from the specification, which we explain in this section.

Allowing unrestricted execution requests comes with the problem that malicious users can send requests whose execution takes a disproportional amount of time, e.g., due to infinite loops. If the execution time exceeded the boundaries defined by the on-chain timeouts, malicious users could exploit

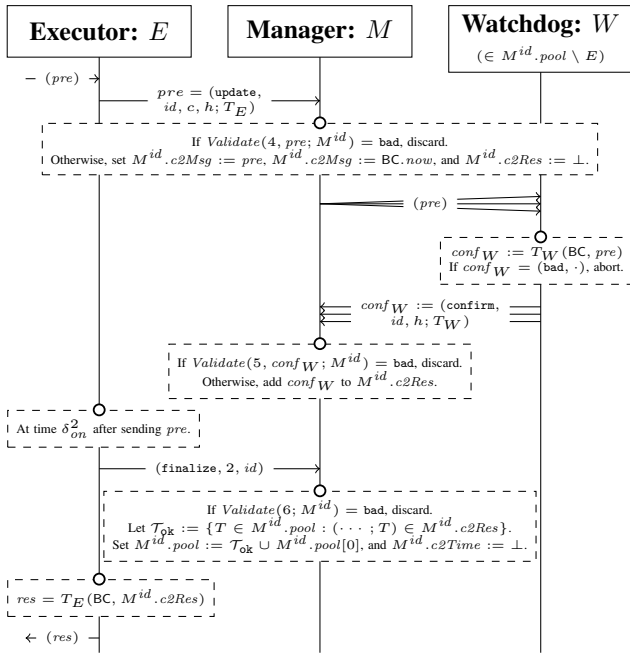


Fig. 4. Detailed watchdog challenge protocol.

this behavior to kick honest operators from an execution pool. This operator *denial of service* attack harms the liveness property of the system. In order to mitigate the vulnerability, we introduce an upper bound to the computation complexity of a single contract execution. Once the bound is reached, the executor enclave stops executing and reverts the state but still provides a valid output. The timeouts in the system are set such that an honest operator cannot be kicked from an execution pool even if an execution takes the maximum amount of computation. The same applies to update and creation requests, where failed creations return a *fail confirmation* that can be submitted to the manager instead of the creation confirmation. A fail confirmation triggers the manager to mark the contract as crashed. Note that the *POSE* system still supports the execution of arbitrary complex smart contracts as the timeouts and hence the upper bounds can be set arbitrarily high (cf. Appendix D). Additionally, all contracts of an operator are executed and challenged independently, and thus, contracts do not block each other.

While we have assumed that all operators run only one *POSE* enclave, multiple enclaves can be created in practice. This enables the opportunity of a *sybil attack*, where a malicious operator generates multiple *POSE* enclaves to increase its share in the system and hence harm the liveness property. This attack can be mitigated by forcing an operator to deposit funds at each enclave registration and which will be paid back to the operator only if she behaves honestly. We note that this deposit is independent of any contract and its parties. Now, such an attack is directly linked to financial loss. See Section VI for more discussions about incentives and fees.

In order to enhance *privacy*, neither users nor operators send inputs or respectively execution results in clear. Instead, users encrypt inputs using hybrid encryption based on the public key of the executor enclave. Additionally, users specify

a symmetric key in their execution request, which is used to encrypt the result of the execution when sent back to the user. This way, inputs and results are private and cannot be eavesdropped by a malicious operator.

The term *griefing* denotes attacks where an adversary forces an honest party to interact with the blockchain in order to generate financial damage to this party. Especially when blockchain transactions require high fees, such attacks pose serious vulnerabilities. In regards to challenges within the *POSE* protocol, we mitigate the attack surface for griefing attacks by incorporating a mechanism in the manager that fairly splits the fees for challenge and response between the challenger and the challenged party. The same mechanism can be used for the contract creation process.

An adversary executing a *clogging* attack sends many transactions to the system to prevent honest users from issuing transactions. In the context of *POSE*, an off-chain clogging attack results in honest clients making an on-chain challenge to ensure that their requests will be processed. Hence, a successful clogging attack has to be performed on-chain. For the on-chain challenge, our system inherits the vulnerabilities of the underlying blockchain.

VI. EXTENSIONS

We simplified some protocol steps in order to make the protocol description more compact and easier to understand. We discuss the most important extensions and their benefits in this section.

Contract & Operator Lifecycle. A mechanism that releases enclaves from their execution duty can be integrated. This allows operators to voluntarily withdraw their enclaves from an execution pool. On the one hand, terminated contracts can be closed, which releases all pool enclaves from their execution duty. On the other hand, it enables to withdraw a single enclave and exchanging it by a randomly chosen replacement enclave. Additionally, a replacement strategy is also applicable to the scenarios in which enclaves are kicked. The latter extension reduces the chance of a contract crash, the event in which no more operator remains. We stress that these extensions can easily be achieved by adding the functionality to our *POSE* program and the manager. In case a contract is idle for a long time, an extension may be implemented that allows operators to *hibernate* their respective enclave. The enclave state can be stored on disk by encrypting it with a key that is kept alive in the hibernating enclave; thus, only requiring minimal overhead in memory. The *POSE* program ensures freshness by synchronizing with the blockchain; thus, preventing rollback attacks.

Incentives. Although *POSE* provides security not only against rational but also byzantine adversaries, it is beneficial to introduce incentives for operators to join the system and act honestly. Moreover, operators can be compensated for on-chain transactions. Such incentives can be achieved by introducing execution fees paid by the users to the operators. We expect these fees to be significantly lower than Ethereum transaction fees since replication of computation is only required among a small pool. Additionally, registration fees for operators can be used to mitigate the risk for sybil attacks. By mitigating these

attacks and due to the random assignment of enclaves to contract pools, operators can only actively enforce centralization at high cost.

Efficiency Improvements. Instead of propagating each contract invocation, a more fine-grained distinction based on the action can be added. In particular, a simple state retrieval must not be propagated. In order to improve the efficiency of the manager, messages and responses are not stored persistently. Instead, only their hashes are stored and the actual data is propagated via events. Moreover, the total on-chain transactions can be reduced by letting the executor enclave challenge only the unresponsive watchdog enclaves.

VII. SECURITY ANALYSIS

In this section, we present security considerations of *POSE* based on the adversary model stated in Section II.

A. Protocol Security

For the sake of brevity, we present the full security analysis of our *POSE* protocol including formal theorems in Appendix A. Here, we provide an intuition of our security guarantees.

The *POSE* protocol satisfies *correctness*, ϵ -*liveness* and *state privacy*.

(1) Intuitively, *correctness* means that an adversary cannot influence the smart contract execution within an enclave such that the result is invalid according to the contract logic. Our creation protocol ensures that all enclaves of a pool store the correct contract code. The TEE security guarantees and the *POSE* code ensure that each enclave executes the stored code correctly. Finally, the synchronization mechanism guarantees that each enclave is up-to-date with the blockchain up to some slack, τ_{slack}^{off} . This ensures that on-chain transactions are considered by the smart contract execution, at least after time τ_{slack}^{off} .

(2) The ϵ -*liveness* property states that every contract execution will eventually be processed with probability ϵ , unless the contract crashes and prevents any further execution. Let n be the number of enclaves in the system, m be the number of malicious enclaves and s be the pool size, then it holds that $\epsilon = 1 - \prod_{i=0}^{s-1} \binom{m-i}{n-i} > 1 - \left(\frac{m}{n}\right)^s$. We achieve these high liveness guarantees by enabling the contract execution to proceed even if only one operator out of a randomly selected pool is honest. Our protocol ensures that honest operators cannot be forced out of the pool.

(3) *State privacy* ensures that an adversary cannot obtain additional information about a contract state besides what she learns from the results of contract executions alone. The integrity guarantees of the TEE protect the state of the contract against the TEE’s operator during computation and at rest. During transit, the state is hidden via encryption. Additionally, our protocol ensures that each contract execution producing an observable result is final. This ensures that the execution cannot be reverted to a state in which a previously published output contains private data that should not have been leaked.

B. Architectural Security

We further examine the architectural security of enclaves. The case of a user or TEE operator going offline by turning off their machine is covered in the protocol security (cf. Section VII-A); here we focus on parties that follow the protocol, trying to gain an unfair advantage in various ways.

The adversary might try to perform a memory corruption attack on the client used by users to interact with the executor (e.g., to send inputs). To mitigate this risk, the software should be implemented in a memory-safe language, like Python or Rust, and be open source so that it can be easily inspected.

A malicious TEE operator can also try mounting a memory-corruption or a side-channel attack on its TEE. As mentioned in A1.1, we assume that the TEE protects the confidentiality of the enclave and prevents leakage. However, in practice, cache-based side-channel attacks have been successfully demonstrated also on ARM processors [44]. While we want to stress that our ARM TrustZone-based implementation is a research prototype and the design is TEE-agnostic, the risk of these attacks can be mitigated by making the TEE opt-out of shared caches and flush private caches upon context switch, as proposed in [19]. Alternatively, a more advanced TEE design can be used [24], [19], [16]. Moreover, if the enclave code has an exploitable memory-corruption vulnerability, it is possible to mount a memory-corruption attack against it. One way to mitigate this risk, and hence, realize our assumption A1.2, is to use a memory-safe language for our smart contracts (in our case, Lua), or to deploy a run-time mitigation (like CFI [11]). Yet, in practice, an adversary might still be able to compromise an enclave. In this case, only the contracts of this enclave are affected. The consequences depend on the role of the enclave: for an executor enclave, the adversary gets full control over the contract; for a watchdog enclave, the adversary can only break state privacy.

Finally, an adversary might build a malicious smart contract with the goal of compromising secrets owned by other contracts or blocking an enclave by entering into an infinite loop. We mitigate against the first scenario by ensuring that only one smart contract is executing at any given time in an enclave, so that no foreign plain text secrets are present in memory at any point during contract execution. In case of multiple enclaves running on the same system, the TEE is isolating enclaves from each other such that no contract can tamper with another (cf. assumption A1.1). To handle infinite loops, we leverage a Lua sandbox [14], which interrupts the execution of the Lua code after a predetermined number of instructions has been issued and disables access to unsafe functions and modules.

VIII. IMPLEMENTATION

In order to evaluate *POSE*, we implemented a prototype for the manager and the enclaves, which uses TrustZone for the enclaves themselves and Lua as the smart contract programming language. We open source our prototype implementation to foster future research in this area⁵. We describe each of them in the following.

Manager. For the manager we use an Ethereum smart contract written in Solidity, which we will refer to as *manager* in the

⁵<https://github.com/AppliedCryptoGroup/PoseCode>

following. Even if this implementation is based on Ethereum, we note that our design can be realized on any blockchain supporting rich smart contracts. The manager keeps a list of all registered enclaves in the network as well as a list of all deployed contracts, including their public information, e.g., the address of the current executor. As mentioned in the protocol described in Section V-E, the manager provides functions to register an enclave, create a new *POSE* contract, deposit or withdraw money, and functions to challenge the current executor or any of the watchdogs. To synchronize all participants, every time a challenge related function was called it will throw an appropriate Solidity event.

Enclaves. The contract creator, executor, and watchdogs are enclaves running in a TEE. As our protocol is TEE-agnostic and all commercial TEEs exceed smart contracts’ on-chain requirements on memory/computational-power capabilities significantly, we chose to use ARM TrustZone [15] for our prototype. TrustZone features a traditional programming model (OS, and user-space applications with standard library), and the Open Portable Trusted Execution Environment (OP-TEE) OS [42] already supports a large fraction of standard functionality, and hence, does not force us to reimplement this for the contract execution environment. TrustZone supports two execution modes: secure world and normal world. The system’s memory can be freely distributed among these worlds. The secure world is an trusted OS which is completely independent from the normal OS, which in our case is Linux. Code running in the secure world is called a *Trusted App* (TA). A TA may only communicate with the normal world via shared memory regions, which are explicitly allocated as such. We implement the *POSE* enclaves as TAs. Computations in the secure world have native performance; yet, switching between worlds has a constant but negligible overhead (in our tests around 449 μ s). TrustZone does not impose memory limits for secure world. While we leverage the traditional TrustZone concept, recent versions add support for a S-EL2 hypervisor to allow multiple strongly isolated enclaves that allows *POSE* to scale better on these platforms. Most basic cryptographic functions are provided by the OP-TEE TA library, such as AES and TLS. Note that TrustZone itself does not standardize a remote attestation implementation itself, but industry [3], [6], [8] and OP-TEE implementations exist⁶. Remote attestation can also be used to prove a certain set of software defenses is active in the enclave. In our prototype, we leveraged OP-TEE’s remote attestation functionality to attest the enclave after setting up the runtime. To leverage this feature, the *POSE* enclave requests a signed attestation report from the attestation PTA (Pseudo Trusted App), essentially a kernel module of the OP-TEE OS in secure world. The keys for signing the attestation report are derived using hardware device information and stored persistently after generation (using Secure Storage, or “Trusted Storage”, as defined by GlobalPlatform’s TEE Internal Core API specification).

To properly interact with the Ethereum-based manager, we also adapted and deployed an Ethereum wallet for embedded devices [13], enabling the enclaves to create ECDSA signatures, Keccak hashes, handle encoding, and create transactions to call the manager. For *POSE* contracts, we use the scripting language Lua [53]. It is a well-established, fast, powerful, yet

simple language written in C. Lua as well as the enclave itself allow arbitrary computation. We ported the Lua interpreter to run inside the TA, by stripping out operations unsupported by the TA, such as file access. After each execution step, the enclave returns to the normal world while keeping the contract’s Lua session alive. When the normal world receives an input from a user, it invokes the TA with these inputs to continue the Lua execution. To update the enclave runtime, different approaches are possible in practice, e.g., the manager could announce an update and all outdated enclaves would shut themselves down after a timeout. Honest operators then would incrementally trigger an enclave replacement during the timeout period.

IX. EVALUATION

This section examines *POSE* regarding complexity and performance. In the following, we will report absolute performance numbers and discuss these in relation to Ethereum itself, but also compare to existing works based on TEEs, namely FastKitten and Bitcontracts. FastKitten has a highly similar set of tested smart contracts, so a comparison can put our numbers in perspective. For Bitcontracts, we reimplemented Quicksort with the same experiment setup. Note, that the smart contracts can still be implemented differently, and the performance and the TEE differ.

Complexity. Running a *POSE* contract in the benign case, i.e., if all involved enclaves respond, requires exactly two blockchain interactions for the setup. Each user of a contract also needs one blockchain interaction each time the user deposits or withdraws money regarding the contract. However, as *POSE* does not require a fixed collateral for the setup, the money transactions do not inherently prevent the contract from execution—except the specific contract demands it. Otherwise, when either the executor or any watchdog fails to respond, each challenge requires two blockchain interactions. The delay incurred by our challenge protocol is dominated by the on-chain transactions. This holds also for other off-chain solutions, e.g., state-channels [46], [26], [22], Plasma [52], [37], Rollups [48], [5] and FastKitten [25]. For instance, the time it takes for an honest executor to kick a watchdog is 325s on average. We discuss timeout parameters and the challenge delay more thoroughly in Appendix D. In the worst-case, a malicious operator does not respond to the off-chain messages but to the challenges in every execution step, which would effectively reduce *POSE*’s execution speed beneath that of the blockchain. However, such an attack requires continuous blockchain interactions from the malicious party and hence entails costs for every execution step (cf. Section IX “Manager”).

Test Setup. We deployed a test setup with our prototype implementation for performance measurements. The test setup consists of five devices. For the enclaves we deployed three Raspberry Pi 3B+ with four cores running at 1.4GHz. These are widely available and cheap devices that support ARM TrustZone. As state updates are small (just the delta to the previous state) and watchdogs receive and process the state updates in parallel, we do not expect an increase of the pool size to significantly influence the evaluation. Further, we used `ganache-cli` (6.10.2) to emulate a Ethereum blockchain in our local network, which runs the Solidity contract that

⁶https://github.com/OP-TEE/optee_os/pull/5025

TABLE I. COST OF EXECUTING THE *POSE* MANAGER. THE USD COSTS WERE ESTIMATED BASED ON THE PRICES (GAS TO GWEI AND ETH TO USD) ON MAY. 8, 2022 [27], [21]. *FOR COMPARISON, THESE ARE THE COSTS OF POPULAR OPERATIONS ON ETHEREUM.

Method	Cost	
	Gas	USD
registerEnclave	175 910	13.23
initCreation	198 436	14.91
finalizeCreation	79 545	5.98
deposit	37 255	2.80
withdraw	36 997	2.78
challengeExecutor	54 654	4.11
executorResponse	51 478	3.87
executorTimeout	53 327	4.01
challengeWatchdogsCreation	231 286	17.38
challengeWatchdog	131 362	9.87
watchdogResponse	36 257	2.72
watchdogTimeout	52 142	3.92
simple Ether transfer*	21 000	1.58
create CryptoKitty*	250 000	18.78

implements the manager. Finally, a fifth device emulates multiple users by simply sending out network requests to both the manager and enclave operators, which are all connected via Ethernet LAN.

Manager. As the *POSE* manager is implemented as an Ethereum smart contract, interactions with it incur some costs in the form of Gas. The costs of all implemented methods of the Solidity contract are listed in Table I. The first five methods are used for benign *POSE* contract execution. The second part of the table shows methods that are required for challenges, including the response and timeout methods to resolve them. In terms of storage, each additionally registered enclave will require 64 bytes and each contract 288 bytes + (pool size \times 32 bytes) of on-chain storage.

Contract Execution. To measure and demonstrate the efficiency of *POSE* contract execution, we implemented three applications as Lua code in our test setup. All time measurements are averaged over 100 runs. Regardless of the used contract, setting up an executor or watchdog enclave with a Lua contract takes 189ms. Creating an attestation report for the enclave takes another 367ms with OP-TEE’s built-in remote attestation using a one-line dummy contract. For our biggest contract, Poker, the attestation takes 377ms, resulting in a total setup time of 566ms. In contrast, FastKitten needs 2s for enclave setup. Note that FastKitten needs an additional blockchain interaction. Multiple contracts run by a single operator are executed in parallel, including network communication. Thus, the number of enclaves, contracts and transactions a single operator can process depends on the operator’s hardware. As modern servers CPUs feature 128 cores [23], and servers often feature multiple CPUs, we do not expect parallel execution to affect performance significantly. However, to prevent overload, the number of pools an operator participates in can be limited.

Rock paper scissors. This is an implementation of the popular game with two players. Unlike traditional smart contracts, we can leverage *POSE*’s private state to simply store each player’s input, instead of having to use much more complex multi-round commitments. The resulting smart contract is 27 lines of code (LoC). Disregarding the delay caused by human players,

the execution time of one round with two user inputs is 32ms. In comparison, FastKitten only needs 12ms, but is also running on a much more powerful machine. In contrast, executing this game on Ethereum would take around 5 minutes for each round (20 confirmation blocks, 15s block time each).

Poker. We have also implemented Poker as a multi-party contract running over multiple rounds. Note that in *POSE*, the poker game can be implemented as an ongoing cash game table, i.e., players may join or leave the table at any time, as contracts in *POSE* do not have to be finite. Each round consists of three phases each requiring an input from all users. The resulting smart contract is 209 lines of code (LoC). We execute the contract with five players who have their deposit ready at the start, with a total execution time of 199ms (vs. 45ms in FastKitten, but again, on a more powerful machine). Playing this game on Ethereum would take 5 minutes per player input.

Federated Machine Learning. For this application, users can submit locally trained models, which will be aggregated to a single model by the contract. Any user can then request the new model from the contract. For our measurements, each user trained a convolutional neural network consisting of 431 080 individual weights on the MNIST handwritten digits dataset [62]. For aggregation, the contract averages every existing weight with the corresponding weight sent by the user. The smart contract itself is only 5 LoCs, as we load the existing weights separately. Each aggregation took 238ms, which demonstrates the efficiency of *POSE*. Trying to execute the same function on Ethereum, for each aggregation, storage of the weights alone would exceed 1 billion gas (assuming 4 bytes float per weight) and the calculation over 3.4 million gas (8 gas per weight).

Quicksort. We have also implemented Quicksort to sort a hardcoded input array of 2048 random integers, as done in Bitcontracts [59]. The resulting smart contract is 29 lines of code (LoC). The total execution time of the contract is 20ms. Compared to the 6ms in Bitcontracts, we use a less powerful machine (Bitcontracts uses an AWS T2.micro instance with a recent Intel processor at 3.3Ghz), while our performance measurement also includes additional steps like context switches and the setup of the enclave runtime. Executing this Quicksort contract on Ethereum would cost around 6.5 million gas.

Watchdog State Updates. When an executor operator has been dropped, a watchdog takes over execution. For this to work, state changes are distributed to the watchdogs. Storing the current state and restoring it on a watchdog takes 17ms for the poker contract (averaged over 100 runs, corrected for network latency), which also has the biggest state among the ones we implemented.

Enclave Teardown. After an executor enclave is not expecting further inputs and finished the smart contract execution, the execution environment has to be cleaned up for the next smart contract, i.e., cryptographic secrets and the smart contract in the shared memory need to be zeroed. This takes 25ms.

X. RELATED WORK

Ethereum [58] is the most prominent decentralized cryptocurrency with support for smart contract execution. However, it is suffering from very high transaction costs and data used by smart contracts is inherently public.

TABLE II. OVERVIEW OF RELATED WORK, n IS #TRANSACTIONS.

	No collateral	Private state	Blockchain interactions (optimistically)	Non-fixed lifetime & group
Ethereum [58]	✓	✗	$O(n)$	✓
MPC [40], [41], [39]	✗	✓	$O(1)$	✗
State Channels [46], [26], [22]	✗	✗	$O(1)$	✗
VM-based [36], [60], [59]	✗	✗	$O(n)$	✓
Ekiden [20]	✗	✓	$O(n)$	✗
FastKitten [25]	✗	✓	$O(1)$	✗
<i>POSE</i>	✓	✓	$O(1)$	✓

Hawk [38] aims for improving the privacy by automatically creating a cryptographic protocol from a high-level program in order to allow computation on private data without disclosing it. However, this complex cryptographic layer further decreases performance of the system and increases costs. Similarly, approaches based on Multiparty Computation (MPC) [40], [41], [39] distribute the computation between multiple parties such that no party can access the cleartext data. These approaches have substantial overhead in performance, communication and collateral required.

One approach to alleviate the complexity limitation are state channels [46], [26], [22], which enable parties to lock some funds on the blockchain, execute complex contracts off-chain, and finally commit the results of the contract to the blockchain. This is efficient if all parties agree on the results; otherwise, the dispute can be solved on-chain, which takes longer and is more expensive.

Arbitrum [36] represents a smart contract as a virtual machine (VM), which is executed privately by a number of “managers”. After execution, if all managers agree on the result of the computation, this result can be simply signed and committed to the blockchain, without the need to perform the computation on chain. In case managers disagree, a bisection algorithm is used to compare subsets of the execution on chain and find which is the first instruction on which the managers disagree, then punish the malicious manager(s). Hence, as long as at least one manager is honest, the correct result is computed. While computationally efficient, this on-chain protocol is still relatively expensive, so Arbitrum also includes financial incentives to encourage the managers to behave. The managers have full access to the VM’s data, so confidentiality is broken if even one manager is malicious. Unlike Arbitrum, *POSE* does not require multiple parties to execute the smart contract: the watchdog enclaves just need to acknowledge the new states, unless the executor enclave fails.

ACE [60] and Bitcontracts [59] are similar to Arbitrum, but they allow the results of contract executions to be approved by a configurable quorum of service providers, not necessarily all of them. Unlike *POSE*, ACE does not support private state and requires on-chain communication per contract invocation. Although the transaction is computed off-chain, the invocation and the result are registered on-chain. Further, Arbitrum and ACE require changes to the blockchain infrastructure, hence, they are harder to deploy in practice.

Ekiden [20] is also an off-chain execution system that leverages TEE-enabled *compute nodes* to perform computation and regular *consensus nodes* that interact with a blockchain. The major drawback of Ekiden is that it requires every computation step to retrieve its initial status from the blockchain, and it only supports input from one client at a time. Moreover, the atomic delivery of the output of each step requires to wait for publication of the updated state before the output is made available to the client. Hence, any highly interactive protocol with multiple participants (e.g., a card game) would incur significant delays between turns just to wait for the blockchain. The paper evaluates on a fast blockchain, Tendermint, but does not quantify its latency for interactive protocols on mainstream blockchains like Ethereum. The Oasis Network uses an updated version of Ekiden [30]; yet, this version still requires to store state on the blockchain after each call.

FastKitten [25] also leverages TEEs to perform off-chain computation. It assumes a rational attacker model, with financial incentives to convince all participants to follow the protocols. If they all do, the communication happens directly between the TEE and them, thus dispensing with the high latency due to blockchain roundtrips. However, FastKitten only supports contracts with a predefined list of participants and a limited lifespan. It also requires the TEE operator to deposit as much as every participant combined as collateral. *POSE* lifts those restrictions: it enables long-lived smart contracts with an unknown set of participants and requires no collateral from the TEE owners. Further, *POSE* achieves strong liveness guarantees in the presence of byzantine adversaries, while FastKitten assumes a rational adversary.

ROTE [45] is an approach to detect rollback attacks on TEEs by storing a counter on other TEEs. This approach is similar to the watchdog enclaves used in *POSE* to ensure that execution of a smart contract continues. However, unlike *POSE*, ROTE can only detect rollback attacks, but cannot prevent malicious operators from withholding the state. SlimChain [61] primarily aims at reducing on-chain storage, while still requiring blockchain interactions to store state commitments. Further, the paper does not address storage nodes crashing, which would lead to a liveness violation. Pointproofs [33] proposes a new vector commitment scheme to reduce the storage requirements on blockchain validators. Although validators do not need to store all values of a smart contract, once a transaction provides these values, the execution is still performed on-chain. In contrast, *POSE* works entirely off-chain in the optimistic case and ensures liveness.

Chainspace [12] proposes an entirely new distributed ledger platform focusing on sharding combined with a directed acyclic graph structure, while *POSE* extends established blockchains (e.g., Ethereum). ResilientDB [54] proposes a consensus protocol that clusters validators’ geo-location to minimize network overheads. In contrast, *POSE* is an off-chain execution protocol for smart contracts. Hyperledger Fabric Private Chaincode [29] requires trust in handling the encryption key by the client or an *admin*; thus, we deem it not applicable to permissionless blockchains, targeted by *POSE*. Hyperledger *Private Data Objects* [18], an alternative to Private Chaincode, requires periodic blockchain interactions to store the state on-chain. This slows execution on contract calls to the speed of the blockchain, unlike *POSE*, which executes contracts entirely

off-chain in the optimistic case. Hyperledger *Avalon* [28] can outsource workloads to TEE enclaves. However, these workloads have to be self-contained, and thus, interactions by participants still require on-chain transactions, while *POSE* can run interactive contracts completely off-chain (e.g., Poker).

XI. CONCLUSION

Smart contracts have become an indispensable tool in the era of blockchains; yet, current approaches suffer from various shortcomings. In this paper, we introduce *POSE*, a novel off-chain execution protocol that addresses all of these shortcomings to enable much more versatile smart contracts. We showed *POSE*'s security and demonstrated its feasibility with a prototype implementation.

ACKNOWLEDGEMENTS

This work was supported by the European Space Operations Centre with the Networking/Partnering Initiative, the German Federal Ministry of Education and Research within *Sanctuary* (16KIS1417) and within the *iBlockchain project* (16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 – 236615297 (CROSSING Project S7)*, by the European Union's Horizon 2020 Research and Innovation program under Grant Agreement No. 952697 (*ASSURED*), by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*.

REFERENCES

- [1] Cardano. <https://cardano.org/>. (Accessed on 05/20/2021).
- [2] Cryptokitties - collect and bread furrever friends! <https://www.cryptokitties.co/>. Accessed 14-08-2022.
- [3] Enhanced attestation (v3). <https://docs.samsungknox.com/dev/knox-attestation/about-attestation.htm>. Accessed 20-04-2022.
- [4] Etherscan - ethereum average block time chart. <https://etherscan.io/chart/blocktime>. Accessed 20-09-2021.
- [5] Optimistic rollups - ethhub. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/optimistic_rollups/. (Accessed on 05/20/2021).
- [6] Qualcomm® trusted execution environment (tee) v5.8 on qualcomm® snapdragon™ 865 security target lite. <https://www.tuv-nederland.nl/assets/files/cerfificaten/2021/08/nsicb-cc-0244671-stlite.pdf>. Accessed 20-04-2022.
- [7] Solidity documentation. <https://docs.soliditylang.org/en/v0.8.7/>. Accessed 20-09-2021.
- [8] Upgrading android attestation: Remote provisioning. <https://android-developers.googleblog.com/2022/03/upgrading-android-attestation-remote.html>. Accessed 20-04-2022.
- [9] Proxy bid. https://en.wikipedia.org/w/index.php?title=Proxy_bid&oldid=968758683, July 2020.
- [10] Google cloud bigquery: Block variance. <https://console.cloud.google.com/bigquery>, 2021. Query: SELECT b.timestamp FROM 'bigquery-public-data.ethereum_blockchain.live_blocks' AS b ORDER BY b.timestamp; Accessed 20-09-2021.
- [11] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. CFI: Principles, implementations, and applications. In *Proc. ACM Conference and Computer and Communications Security (CCS)*, 2005.
- [12] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hryczyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *25th Annual Network and Distributed System Security Symposium, (NDSS 2018)*, 2018.
- [13] AnyLedger. Embedded Ethereum wallet library GitHub. <https://github.com/AnyLsite/embedded-ethereum-wallet>, 2020.
- [14] APItools. sandbox.lua. <https://github.com/APItools/sandbox.lua>, 2017.
- [15] ARM Limited. ARM Security Technology: Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2008.
- [16] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stempf. CURE: A security architecture with Customizable and Resilient Enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [17] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1521–1538, 2019.
- [18] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. Private data objects: an overview. *CoRR*, abs/1807.05686, 2018.
- [19] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stempf. SANCTUARY: ARMing TrustZone with user-space enclaves. In *NDSS*, 2019.
- [20] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.
- [21] CoinMarketCap. Ethereum (ETH) price. <https://coinmarketcap.com/currencies/ethereum/>, 2020.
- [22] Jeff Coleman, Liam Horne, and Li Xuanji. Counterfactual: Generalized state channels, Jun 2018. <https://l4.ventures/papers/statechannels.pdf>.
- [23] Ampere Computing. Ampere Altra Max 64-Bit Multi-Core Processor Features. <https://amperecomputing.com/processors/ampere-altra/>, 2022.
- [24] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.
- [25] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. Fastkitten: practical smart contracts on bitcoin. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.
- [26] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018.
- [27] Etherscan. Ethereum Average Gas Price Chart. <https://etherscan.io/chart/gasprice>, 2020.
- [28] Hyperledger Foundation. Hyperledger avalon. <https://wiki.hyperledger.org/display/avalon/Hyperledger+Avalon>. Accessed 04-08-2022.
- [29] Hyperledger Foundation. Hyperledger fabric private chaincode. <https://github.com/hyperledger/fabric-private-chaincode>. Accessed 04-08-2022.
- [30] Oasis Foundation. An implementation of ekiden on the oasis network. <https://oasisprotocol.org/papers>. Accessed 04-08-2022.
- [31] Tommaso Frassetto, Patrick Jauernig, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi. POSE: Practical off-chain smart contract execution. *CoRR*, abs/2210.07110, 2022.
- [32] GlobalPlatform. TEE Internal Core API Specification. <https://globalplatform.org/specs-library/tee-internal-core-api-specification-v1-2/>, 2019.
- [33] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2007–2023, 2020.
- [34] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.
- [35] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: Epid provisioning and attestation services. *White Paper*, 1(1-10):119, 2016.
- [36] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart

- contracts. In *27th USENIX Security Symposium (USENIX Security 2018)*. USENIX Association, 2018.
- [37] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. Commit-chains: Secure, scalable off-chain payments. *Cryptology ePrint Archive, Report 2018/642*, 2018.
- [38] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.
- [39] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [40] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.
- [41] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [42] Linaro, Inc. OP-TEE Documentation. <https://readthedocs.org/projects/optee/downloads/pdf/latest/>, 2020.
- [43] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Peter Pietzuch, and Emin Gün Sirer. Teechain: Reducing storage costs on the blockchain with offline payment channels. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 125–125, 2018.
- [44] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.
- [45] Sinisa Matetic, Mansoor Ahmed, Kari Kostiaainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.
- [46] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR*, abs/1702.05812, 2017.
- [47] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2008.
- [48] Offchain Labs, Inc. Arbitrum rollup: Off-chain contracts with on-chain security. 2020.
- [49] Gustavo A Oliva, Ahmed E Hassan, and Zhen Ming Jack Jiang. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empirical Software Engineering*, 2020.
- [50] Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2017.
- [51] Travis Patron. What’s the big idea behind Ethereum’s world computer. <https://www.coindesk.com/whats-big-idea-behind-ethereums-world-computer/>, 2016.
- [52] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. 2017.
- [53] PUC-Rio. The programming language Lua. <https://www.lua.org/>, 2020.
- [54] Sajjad Rahnama, Suyash Gupta, Thamir M Qadah, Jelle Hellings, and Mohammad Sadoghi. Scalable, resilient, and configurable permissioned blockchain fabric. *Proceedings of the VLDB Endowment*, 13(12), 2020.
- [55] Andrey Sergeenkov. How to check your ethereum transaction. <https://www.coindesk.com/learn/how-to-check-your-ethereum-transaction/>. Accessed 24-08-2022.
- [56] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.
- [57] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *CoRR*, abs/1908.04756, 2019.
- [58] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.
- [59] Karl Wüst, Loris Diana, Kari Kostiaainen, Ghassan Karame, Sinisa Matetic, and Srdjan Capkun. Bitcontracts: Adding expressive smart contracts to legacy cryptocurrencies. 2019.
- [60] Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiaainen, and Srdjan Capkun. ACE: asynchronous and concurrent execution of complex smart contracts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.
- [61] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. Slimchain: scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment*, 14(11):2314–2326, 2021.
- [62] Yann LeCun and Corinna Cortes and Christopher J.C. Burges. THE MNIST DATABASE. <http://yann.lecun.com/exdb/mnist/>, 2020.
- [63] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 270–282, 2016.
- [64] Fan Zhang, Philip Daian, Iddo Bentov, and Ari Juels. Paralysis proofs: Safe access-structure updates for cryptocurrencies and more. *IACR Cryptol. ePrint Arch.*, 2018:96, 2018.

APPENDIX

A. Protocol Security

We analyze the security of our protocol under the assumption of an IND-CPA secure encryption scheme, an EU-CMA secure signature scheme and a collision resistant hash function in the following. We present definitions of correctness, ϵ -liveness and state privacy.

1) Correctness: We define a state update as the evaluation of a transition function f , which receives as inputs a user U , a user input $move$ and a copy of the blockchain BC. The *correctness* property states that each state update evaluates the transition function as defined by the contract code with valid inputs, i.e., U is the (potentially malicious) client triggering the transition, $move$ the input of U and BC a valid copy of the blockchain that is at most τ_{slack}^{off} behind the main chain.

Claim 1 (Correctness): *POSE satisfies correctness.*

We first note that according to our adversary model, a corrupted operator may delete any message intended for her enclave or generated from her enclave. However, the correct execution of the *POSE* program inside the enclave cannot be influenced. When an operator creates a *POSE* enclave, the registration process ensures that the new enclave indeed runs the *POSE* program. To this end, our protocol utilizes the TEE attestation mechanism, which generates a verifiable statement that the enclave is running a specific program. Upon registration with the manager M , M checks the validity of the attestation statement as well as the blockchain evidence, the signed hash and number of the latest block known to the enclave. M only registers the enclave in the system if the new enclave is running the *POSE* program and is not further behind than maximally τ_{slack}^{off} . Finally, the TEE integrity and confidentiality guarantees ensure that a malicious operator cannot modify the enclave’s code, tamper with its state or access its private data, in particular, its signature keys.

During the creation of a contract, the pool enclaves attest the code of the installed contract to the creation enclave. The creator checks that the code is consistent with the hash stored in the manager before signing a creation confirmation. Hence, it is not possible, without breaking the EU-CMA security of the signature scheme or the collision resistance of the hash function, to create a valid creation confirmation for a contract with different code than specified by the creation request.

Next, contract state updates can only be triggered by invoking the executor enclave with an execution request or invoking a watchdog enclave with an update request. The correctness of the latter is reduced to the correctness of the former. To see this, we observe that any update request to a watchdog enclave requires to be signed by the executor enclave. Clearly, the executor enclave only signs updates corresponding to its own executions. Therefore, an adversary cannot forge incorrect update request without breaking the unforgeability of the signature scheme. Also, the executor enclave can only issue a new state update if all watchdogs confirmed the previous one. Hence, it is not possible to tamper with the order in which the update requests are provided to a watchdog enclave. As stated before, the TEE integrity guarantees ensure the correct execution of the program code and hence the correct execution of the smart contract. It follows that a state update can only be achieved by providing inputs to the executor enclave. The executor enclave receives a signed message containing the action *move* from user U and the relevant blockchain data from its operator. In Section V-D, we describe how our protocol achieves secure synchronization between the executor enclave and the blockchain. In particular, the synchronization mechanism ensures that the blockchain data accepted by an enclave is correct and complete in regard to a correct blockchain copy that is at most τ_{slack}^{off} behind the main chain. This guarantees that BC, represented by the received blockchain data, is a synchronized copy of the current blockchain. In order to protect inputs by honest users U , *move* needs to be signed by U . This means an adversary cannot tamper with the input without breaking the signature scheme.

Finally, we note that each *POSE* enclave maintains a list of received messages. Since an honest user randomly selects a fresh nonce for each execution request, replay attacks can be detected and prevented by any executor enclave.

2) *Liveness*: The liveness property states that every contract execution initiated by an honest user U will eventually be processed with high probability. For a successful execution, a valid execution response is given by the executor. Unsuccessful execution can only happen in case of a contract *crash*. In this event, the contract execution halts and neither honest nor malicious users can perform successful contract executions anymore. We emphasize that the pool size can be set such that crashes happen only with negligible probability. In particular, for ϵ -liveness, the probability of a crash is bounded by $1 - \epsilon$.

Claim 2 (ϵ -Liveness): *Let n be the total number of enclaves in the system, m be the number of malicious operators' enclaves and s be the contract pool size. *POSE* satisfies ϵ -liveness for $\epsilon = 1 - \prod_{i=0}^{s-1} (\frac{m-i}{n-i}) > 1 - (\frac{m}{n})^s$.*

Whenever user U sends an execution request to the executor enclave \mathcal{E}_E , U either directly receives a response or U challenges \mathcal{E}_E via the manager M . If \mathcal{E}_E does not respond within some predefined timeout, it will be kicked out of the execution pool and one of the watchdog enclaves takes over the executor role. User U can now trigger the execution again by interacting with the new executor enclave. During execution, the executor enclave \mathcal{E}_E requires confirmations from all watchdog enclaves in order to produce a valid result. However, watchdog enclaves cannot stall the execution forever, as \mathcal{E}_E is able to challenge them via the manager. All unresponsive watchdog enclaves will

be kicked out of the execution pool—the confirmations from the remaining watchdogs suffice to create a result. We stress that all timeouts are defined in Appendix D with great care to ensure that honest operators have enough time to respond. For example, the timeout for the executor challenge is sufficient to allow the executor enclave to challenge the watchdog enclaves twice; once for a currently running off-chain execution and once for the challenged on-chain execution. Although *POSE* guarantees that honest operators' enclaves will never be kicked, there is a small probability that an execution pool consists only of malicious operators' enclaves. If all enclaves are kicked out of the execution pool, the contract execution crashes. Let n be the number of total registered enclaves, m denote the number of enclaves controlled by malicious operators, and s the execution pool size. The probability of a crash is equal to the probability that only malicious operators' enclaves are within an execution pool. This is bounded by $\epsilon = 1 - \prod_{i=0}^{s-1} (\frac{m-i}{n-i}) > 1 - (\frac{m}{n})^s$. Hence, *POSE* achieves ϵ -liveness.

Assuming a total of $n = 100$ registered enclaves and $m = 70$ of them are controlled by malicious operators. Even in this setting with a large share of malicious operators, *POSE* achieves liveness with $\epsilon > 92\%$ for a pool size of just 7. If only half of the operators are malicious, i.e., $m = 50$, *POSE* achieves liveness with $\epsilon > 99\%$ for the same pool size of 7. For $m = 10$ malicious operators, a pool size of only 3 yields a liveness with $\epsilon > 99\%$. For the same scenario of 10% malicious operators and assuming 40 millions contracts running in *POSE*, the pool size of 11 results in a probability of more than 99% that there is no crash at all in the whole system. See Fig. 5 for an illustration of the probability of no crashes depending on the number of contracts for different pool sizes.

3) *State Privacy*: The *state privacy* property says that the adversary cannot obtain additional information about a contract state besides what she learns from the results of contract executions alone.

Claim 3 (State Privacy): *POSE satisfies state privacy.*

The smart contract's state is maintained by the enclaves within the execution pool. According to our adversary model (see Section II), the TEE provides confidentiality guarantees, i.e., the execution of an enclave does not leak any data. Hence, the smart contract's state is hidden from the adversary, even if the enclave's operator is corrupted. The only point in time when information about the contract's state is revealed is at the end of the execution protocol. However, the data provided as a result contains only public state and hence does not reveal anything about the private state. During the execution protocol, the executor enclave propagates the new state to all watchdog enclaves. However, the transferred data is encrypted using an IND-CPA secure encryption scheme. The security of the scheme guarantees that an adversary seeing the message cannot extract information from it. While an enclave only publishes outputs after successful executions, we need to show that each produced output is final. In particular, a succeeding executor must not be able to revert to a state in which a published output should not have been produced. To this end, the state of the executor enclave producing a particular output needs to be replicated among all other enclaves before revealing the actual output. This property is achieved by the state propagation

mechanism of *POSE*. An enclave only returns an output if all enclaves in the pool confirm the corresponding state update. The EU-CMA secure signature scheme guarantees unforgeability of the confirmations. Hence, each confirmation guarantees that the corresponding enclave has updated its state correctly. Further, the correctness property of our protocol (cf. Section A1) ensures an enclave is always executed with a correct blockchain copy; thus, is always aware of the correct pool composition. This means an output can only be returned if the whole pool received the corresponding state update.

B. Supported Contracts

POSE supports contracts with a dynamic set of users of arbitrary size and an unrestricted lifetime. The timeouts need to be set reasonable with respect to the expected execution time of the contracts to allow the execution of complex contracts and to prevent denial of service attacks at the same time. Interaction between *POSE* contracts can be realized by letting the TEE of the calling contract instruct its operator to request an execution of the second contract via the respective executive operator and wait for the response. We deem the exact specification, e.g., enforce an upper bound on (potentially recursive) external calls to guarantee timely request termination, an engineering effort. Calls from *POSE* contracts to on-chain contracts can be supported similarly to our payout concept (Appendix E).

C. Further Protocol Blocks

To keep the specification of the *POSE* protocol in the main body simple and compact, we have excluded the formal specification of the creation process and the validation algorithms. In this section, we present the validation algorithms. For the formal specification of the creation process, we refer the reader to the full version of the paper [31].

All of the different messages sent to the manager throughout the protocol need to be validated with several checks. In order to keep the description compact, we did not include the validation steps in the protocol figures but extracted them into a validation algorithm specified in Program 3. The algorithm is invoked with an counter specifying the checks that should be performed, an optional message that should be checked and the contract state tuple maintained by the manager. The validation returns ok if all requirements are satisfied and M can continue executing and bad if M should discard the received request.

D. Timeouts

Our protocol incorporates several timeouts δ_{off}^* , which define until when an honest user or operator expects a response to a request, and δ_{on}^* , which define until when the manager expects a response to a challenge. These timeouts have to be selected carefully s.t. each honest party has the chance to answer each message and challenge before the respective timeout expires. In this section, we elaborate on the requirements on the timeouts. We neglect message transmission delays and also assume that each challenge sent to the manager will directly be received by all operators (already before it is included into a final block)⁷. We recall the maximum blockchain delay which is defined as $\delta_{BC} = \alpha \cdot \tau$ (cf. II and IV). The off-chain

Program 3: Algorithm *Validate*

The validation algorithm performs the following checks. If input $C = \perp$, the parsing of a message fails or any require is not satisfied, the algorithm outputs bad. Otherwise, it outputs ok.

- On input $(1, m; C)$, parse m to $(\text{execute}, id, \cdot, \cdot; U)$. Require that $C.creator = \perp$, $C.c1Time = \perp$ and $Verify(m) = \text{ok}$.
- On input $(2, res; C)$, parse res to $(\text{ok}, id, \cdot, h; T)$. Require that $C.creator = \perp$, $H(C.c1Msg) = h$, $C.c1Time + \delta_{on}^1 > BC.now$, $Verify(res) = \text{ok}$ and $C.pool[0] = T$.
- On input $(3; C)$, require that $C \neq \perp$, $C.creator = \perp$, $C.c1Msg \neq \perp$ and $C.c1Time + \delta_{on}^1 \leq BC.now$.
- On input $(4, pre; C)$, parse pre to $(\text{update}, id, c, h; T)$. Require that $C.creator = \perp$, $C.c2Time = \perp$, $C.pool[0] = T$ and $Verify(pre) = \text{ok}$.
- On input $(5, conf; C)$, parse $conf$ to $(\text{confirm}, id, h; T_i)$ and $C.c2Msg$ to $(\cdot, \cdot, \cdot, h'; \cdot)$. Require that $C.creator = \perp$, $C.c2Time + \delta_{on}^2 > BC.now$, $Verify(conf) = \text{ok}$, $h = h'$ and $T \in C.pool$.
- On input $(6; C)$, require that $C.creator = \perp$, $C.c2Time \neq \perp$ and $C.c2Time + \delta_{on}^2 \leq BC.now$.

propagation timeout δ_{off}^2 describes the time an execution or creation operator maximally waits for a confirmation from the (other) pool members. It needs to be larger than the maximal update respectively installation time of a contract. Timeout $\delta_{on}^2 \geq \delta_{off}^2 + \delta_{BC}$ describes the maximal time after which M expects a response to any watchdog challenge, either during creation or execution. The off-chain execution timeout δ_{off}^1 describes the maximal time a user waits for a response to an execution request. Note that there might be a running execution and both running and new execution might require a watchdog challenge. In case watchdogs are dropped in the process of such a challenge, the executor needs to be able to notify its enclave about the new pool constellation, and hence, wait until the finalization of the challenge is within a final block. This takes additional time $\Delta = \tau \cdot \gamma$ (cf. IV). Hence, δ_{off}^1 needs to be high enough to enable the challenged executor to perform two contract executions and run two watchdog challenges each taking up to time $\delta_{on}^2 + \delta_{BC} + \Delta$. We elaborate on maximal execution, update, and installation times of contracts in Section V-F. Finally, $\delta_{on}^1 \geq \delta_{off}^1 + \delta_{BC}$ defines the maximal time after which M expects a response to an execution challenge. As the creation is comparable to the execution, we set the timeouts for off-chain creation and creation-challenge accordingly. The timeouts are the upper bound of the delay that can be enforced by malicious operators by withholding messages. To decrease the delays in practice, our implementation incorporates dynamic timeouts. Such a timeout is initially set to match an optimistic scenario where all operators answer directly. Only if the executor signals that a watchdog is not responding, the timeout is increased. For example, δ_{on}^1 is initially set by the manager just high enough to allow the executor to perform the execution offline and to send one on-chain transaction. This on-chain transaction is either the response or a watchdog challenge. In case the executor creates a watchdog challenge, this triggers the manager to increase the δ_{on}^1 timeout for the executor. Similarly, the timeout δ_{on}^2 is increased by the manager if any watchdog is not responding and the executor sends a transaction that kicks this watchdog. The increased timeout allows the executor to provide the kick transaction together with enough confirmation blocks to

⁷We could also add twice the max. message delay to each off-chain timeout and the blockchain confirmation time $\Delta = \tau \cdot \gamma$ to each on-chain timeout.

its enclave to finalize the execution. This dynamic timeout mechanism still allows the executor to respond in time even if a watchdog is not responding, but at the same prevents the executor to stall execution to the maximum although the watchdogs have already responded. While the executor still can create a watchdog challenge to increase the delay, this attack is costly as the executor needs to pay for the on-chain transaction. The value of the off-chain timeout δ_{off}^1 is handled similarly. The client only needs to account for watchdog challenges in the previous execution if there is a running on-chain challenge. If there are no running challenges, a client can decrease δ_{off}^1 to δ_{BC} plus two times the time for the TEE to execute and update a contract. If the executor is unresponsive, the client submits its executor challenge much earlier. We give a concrete evaluation for the case of Ethereum, as this is the platform on which our implementation works. Let $\alpha = 20$ be the number of blocks until a transaction is included in the blockchain in the worst case, and $\alpha_{avg} = 10$ in the average case. Further, we consider the block creation time to be $\tau = 44s$ per block in the worst case and $\tau_{avg} = 15s$ in the average case⁸. Finally, we assume that blocks are final, when they are confirmed by $\gamma = 15$ successive blocks. Since the network delay and the computation time of enclaves are at most just a few seconds, which is insignificant compared to the time it requires to post on-chain transactions, we neglect these numbers for simplicity in the following example. In case the executor (resp. a watchdog) is not responding, it is challenged by the client (resp. the executor). The creation of such a challenge takes $\alpha_{avg} \cdot \tau_{avg} = 150s$ on average. In what follows, due to the dynamic timeout mechanism, the on-chain timeout for both, executor challenge (δ_{on}^1) and watchdog challenge (δ_{on}^2), is initially set to $\alpha \cdot \tau = 880s$. For on-chain timeouts, we need to consider the worst-case parameters to allow honest operators to respond timely in every situation. While a dishonest operator can delay up to the defined timeout, an honest operator responds, and hence, finalizes the challenge in $150s$ on average. In case the challenged operator gets kicked, the (next) executor enclave needs to provide the kick transaction together with enough confirmation blocks to its enclave to finalize the execution. This takes $(\alpha_{avg} + \gamma) \cdot \tau_{avg} = 375s$ on average. For executor challenges, it can happen that the executor submits a watchdog challenge during the timeout period. In this case, which can happen at most twice, the timeout is increased by $880s$. If the challenged watchdog does not reply, and consequently is kicked from the pool, the timeout is increased by $(\alpha + \gamma) \cdot \tau = 1540s$. Note, this worst case is very costly to provoke, and in the general case, an honest executor can finalize the kick of the watchdog in $375s$.

E. Coin Flow

The POSE protocol supports the off-chain execution of smart contracts that deal with coins, e.g., games with monetary stakes. To this end, we provide means to send coins to and receive coins from a contract. In this section, we explain the mechanisms that enable the transfer of money and the intended coin flow of POSE contracts. In order to deposit money to a

⁸For setting α and α_{avg} , we consider a transaction to be included into the blockchain after at most 20 resp. 10 blocks according to [55]. To determine τ , we analyzed the Ethereum history via Google-BigQuery and identified that since 2018 every interval of 20 blocks took at most 44s per block. For τ_{avg} , we take the avg. parameter for Ethereum (cf. <https://etherscan.io/chart/blocktime>).

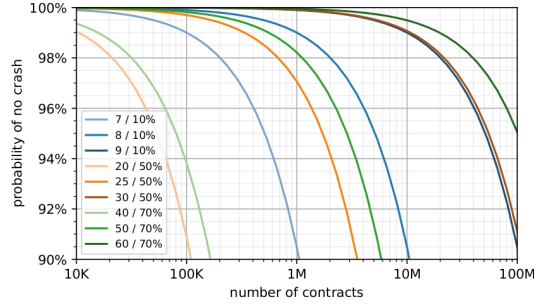


Fig. 5. Cumulative probabilities of no contracts crashing w. large number of POSE contracts for different pool sizes s and adversary shares m , “ s/m ”.

POSE contract, identified by id , a user U sends a message (deposit, id , amount; U) with $amount$ coins to M . Upon receiving a deposit message, M checks whether a contract with identifier id exists and validates the signature, i.e., $M^{id} \neq \perp$ and $Verify(\text{deposit}, id, amount; U) = \text{ok}$. If the checks hold, M increases the contract balance $M^{id}.balance$ by $amount$. As deposits are part of blockchain data that are provided by the operator to an enclave (cf. V-D) and the enclave forwards the data to the $nextState$ function of the contract C^{id} , U is ensured that C^{id} processes the deposit once the corresponding block is final. However, it is upon to the application logic to decide how deposits are processed. A contract C can transfer coins to users by outputting $withdrawals$ as part of the public state. It is upon the application logic to decide how and when coins are transferred to the users. For example, a game can issue withdrawals once the winner has been determined or leave the coins locked for another round unless a user explicitly requests a withdrawal via a contract execution. However, once a withdrawal has been issued, the coins are irreversible transferred. Technically, contract C with identifier id maintains a list of all unspent withdrawals $\{amount_i, U_i\}$ and a counter $payouts$ for the number of spent payouts. Each public state returned by C contains a payout, a signed message $m := (\text{withdraw}, id, payouts, \{amount_i, U_i\}; \mathcal{E}_E)$ where \mathcal{E}_E is the executor enclave of the contract. This message can be sent to M to spent all withdrawals within the payout. M checks the validity of the payout, i.e., $Verify(m) = \text{ok}$, $\mathcal{E}_E = M^{id}.pool[0]$, and $payouts = M^{id}.payouts$. If the checks hold, M transfers coins to the users according to the withdrawal list $\{amount_i, U_i\}$. Finally, M sets $M^{id}.payouts := payouts + 1$ and $M^{id}.balance := M^{id}.balance - sum$, where sum is the sum of all withdrawals. Once C processes a final block with a payout transaction, it updates its list of unspent withdrawals $\{amount_i, U_i\}$ accordingly and increments $payouts$ by 1. This mechanism ensures that a malicious user can neither double spent withdrawals nor prevent an honest user from withdrawing his coins—as long as the contract remains live. Note that for each value of $payouts$, only one payout can be submitted successfully, and a contract only issues a payout for the next value of $payouts$ once it has processed a final block containing the current value of $payouts$. As the contract removes already spent withdrawals from the list, double-spending of any withdrawal is prevented. Although a payout temporarily invalidates all other payouts for the same $payouts$, and hence, might invalidate same withdrawals, the unspent withdrawals will be included in each payout of the incremented $payouts$ and spent with the next payout submission.

F. Prisma : A Tierless Language for Enforcing Contract-client Protocols in Decentralized Applications

This chapter corresponds to the following publication. The full version is available at [153].

- [152] D. Richter, D. Kretzler, P. Weisenburger, G. Salvaneschi, S. Faust, and M. Mezini. “Prisma : A Tierless Language for Enforcing Contract-client Protocols in Decentralized Applications”. In: *ACM Trans. Program. Lang. Syst.* 3 (2023), 17:1–17:41. **Part of this thesis.**

Prisma: A Tierless Language for Enforcing Contract-Client Protocols in Decentralized Applications

DAVID RICHTER and DAVID KRETZLER, Technical University of Darmstadt, Germany
PASCAL WEISENBURGER and GUIDO SALVANESCHI, University of St. Gallen, Switzerland
SEBASTIAN FAUST and MIRA MEZINI, Technical University of Darmstadt, Germany

Decentralized applications (dApps) consist of smart contracts that run on blockchains and clients that model collaborating parties. dApps are used to model financial and legal business functionality. Today, contracts and clients are written as separate programs – in different programming languages – communicating via send and receive operations. This makes distributed program flow awkward to express and reason about, increasing the potential for mismatches in the client-contract interface, which can be exploited by malicious clients, potentially leading to huge financial losses.

In this paper, we present Prisma, a language for tierless decentralized applications, where the contract and its clients are defined in one unit and pairs of send and receive actions that “belong together” are encapsulated into a single direct-style operation, which is executed differently by sending and receiving parties. This enables expressing distributed program flow via standard control flow and renders mismatching communication impossible. We prove formally that our compiler preserves program behavior in presence of an attacker controlling the client code. We systematically compare Prisma with mainstream and advanced programming models for dApps and provide empirical evidence for its expressiveness and performance.

CCS Concepts: • **Software and its engineering** → **Distributed programming languages**; Domain specific languages; Compilers.

Additional Key Words and Phrases: Domain Specific Languages, Smart Contracts, Scala

ACM Reference Format:

David Richter, David Kretzler, Pascal Weisenburger, Guido Salvaneschi, Sebastian Faust, and Mira Mezini. 2022. Prisma: A Tierless Language for Enforcing Contract-Client Protocols in Decentralized Applications. *ACM Trans. Program. Lang. Syst.* 1, ECOOP, Article 1 (January 2022), 44 pages.

1 INTRODUCTION

dApps enable multiple parties sharing state to jointly execute functionality according to a predefined agreement. This predefined agreement is called a *smart contract* and regulates the interaction between the dApp’s clients. Such client–contract interactions can be logically described by state machines [55, 56, 79, 84] specifying which party is allowed to do what and when.

dApps can operate without centralized trusted intermediaries by relying on a blockchain and its consensus protocol. To this end, a contract is deployed to and executed on the blockchain, which guarantees its correct execution; clients that run outside of the blockchain can interact with the contract via transactions. A key feature of dApps is that they can directly link application logic with transfer of monetary assets. This enables a wide range of correctness/security-sensitive business

Authors’ addresses: [David Richter](mailto:david.richter@tu-darmstadt.de), david.richter@tu-darmstadt.de; [David Kretzler](mailto:david.kretzler@tu-darmstadt.de), david.kretzler@tu-darmstadt.de, Technical University of Darmstadt, Hochschulstr. 10, 2052, 64289, Germany; [Pascal Weisenburger](mailto:pascal.weisenburger@unisg.ch), pascal.weisenburger@unisg.ch; [Guido Salvaneschi](mailto:guido.salvaneschi@unisg.ch), guido.salvaneschi@unisg.ch, University of St. Gallen, Torstrasse 25, 9000, St. Gallen, Switzerland; [Sebastian Faust](mailto:sebastian.faust@tu-darmstadt.de), sebastian.faust@tu-darmstadt.de; [Mira Mezini](mailto:mezini@informatik.tu-darmstadt.de), mezini@informatik.tu-darmstadt.de, Technical University of Darmstadt, Pankratiusstraße 2, 2052, 64289, Germany.

2022. 0164-0925/2022/1-ART1 \$15.00
<https://doi.org/>

applications, e.g., for cryptocurrencies, crowdfunding, and public offerings,¹ and the same feature makes them an attractive target for attackers. The attack surface is wide since contracts can be called by any client in the network, including malicious ones that try to force the contract to deviate from the intended behavior [36]. In recent years, there have been several large attacks exploiting flawed program flow control in smart contracts. Most famously, attackers managed to steal around 50 M USD [23, 36] from a decentralized autonomous organization, the DAO. In two attacks on the Parity multi-signature wallet, attackers stole cryptocurrencies worth 30 M USD [12] and froze 150 M USD [65].

Programming dApps. In this paper, we explore a programming model that ensures the correctness and security of the client–contract interaction of dApps by-design. Deviations from the intended interaction protocols due to implementation errors and/or malicious attacks are a critical threat (besides other issues such as arithmetic or buffer overflows, etc.) as demonstrated e.g., by the DAO attack [23, 36] mentioned above.

dApps are multi-party applications. For such applications, there are two options for the programming model: a *local* and a *global model*. In a *local model*, parties are defined each in a separate *local* program and their interactions are encoded via effectful send and receive instructions. Approaches that follow this model stem from process calculi [46] and include actor systems [2] and approaches using session types [27], and linear logic [86]. In contrast, in a *global model*, there is a single program shared by both parties and interactions are encoded via combined send-and-receive operations with no effects visible to the outside world. This model is represented by tierless [16, 22, 38, 69, 70, 80, 81, 87] and choreographic [40, 47, 59] languages. The local model requires an explicitly specified protocol to ensure that every send effect has a corresponding receive operation in an interacting – separately defined – process. With a global model, there is no need to separately specify such a protocol. All parties run the same program in lock-step, where a single send-and-receive operation performs a send when executed by one party and a receive by the other party. Due to encapsulating communication effects, there is no non-local information to track – the program’s control flow defines the correct interaction and a simple type system is sufficient.

Current approaches to dApp programming – industrial or research ones – follow a local model, Contract and client are implemented in separate programs, thus safety relies on explicitly specifying the client–contract interaction protocol. Moreover, the contract and clients are implemented in different languages, hence, developers have to master two technology stacks.

The dominating approach in industry uses Solidity [58] for the contract and JavaScript for clients. Solidity relies on developers following best practices recommending to express the protocol as runtime assertions integrated into the contract code [33]. Failing to correctly introduce assertions may give parties illegal access to monetary values to the detriment of others [52, 60].

The currently dominant encoding style of the protocol as *finite state machine (FSM)* uses one contract-side function per FSM transition [18–20, 58, 75, 76]. While FSMs model a useful class of programs that can be efficiently verified, writing programs in such style directly has several shortcomings. First, an FSM corresponds to a control-flow graph of basic blocks, which is low-level and more suited as an internal compiler representation than as a front-end language for humans. Second, with the FSM style, the contract is a passive entity whose execution is driven by clients. This design puts the burden of enforcing the protocol on the programmers of the contract, as they have to explicitly consider in what state which messages are valid and reject all invalid messages from the clients. Otherwise, malicious clients would be able to force the contract to deviate from

¹700 K to 2.7 M contracts have been deployed per month between July 2020 and June 2021 [43] on the Ethereum blockchain – the most popular dApps platform [24]. Some dApps manage tremendous amounts of assets, e.g., Uniswap [85] – the largest Ethereum trading platform had a daily trading volume of 0.5 B – 1.5 B USD in June 2021.

its intended behavior by sending messages that are invalid in the current state. Third, ensuring protocol compliance statically to guarantee safety requires advanced types, as the type of the next action depends on the current state.

In research, some smart contract languages [9, 18–20, 25, 61, 75, 76] have been proposed to overcome the FSM-style shortcomings. They rely on advanced type systems such as session types, type states, and linear types. There, processes are typed by the protocol (of side-effects such as sending and receiving) that they follow and non-compliant processes are rejected by the type-checker.

The global model has not been explored for dApp programming – which is unfortunate given the potential to get by with a standard typing discipline and to avoid intricacies and potential mismatches of a two-language stack. Our work fills this gap by proposing Prisma – the first language that features a *global programming model* for Ethereum dApps. While we focus on the Ethereum blockchain, we believe our techniques to be applicable to other smart contract platforms as well.

Prisma. Prisma enables interleaving contract and client logic within the same program and adopts a *direct style (DS)* notation for encoding send-and-receive operations akin to languages with baked-in support for asynchronous interactions, e.g., via `async/await` [8, 73]. Prisma leaves it to the compiler to map down high-level declarative DS to low-level FSM style. It avoids the need for advanced typing discipline and allows the contract to actively ask clients for input, promoting an execution model where a dominant acting role controls the execution and diverts control to other parties when their input is needed, which matches well the dApp setting.

Overall, Prisma relieves the developer from the responsibility of correctly managing distributed, asynchronous program flows and the heterogeneous technology stack. Instead, the burden is on the compiler, which distributes the program flow by means of selective continuation-passing-style (CPS) translation and defunctionalisation, as well as inserts guards against malicious client interactions.

For this, we needed to develop a CPS translation for the code that runs on the Ethereum Virtual Machine (EVM), since the EVM has no built-in support for concurrency primitives to suspend execution and resume later – which could be used, otherwise, to implement asynchronous communication. Given that CPS translations reify control flow, without proper guarding, malicious clients could force the contract to deviate from the intended flow by passing a spoofed value to the contract. Thus, it is imperative to prove that our *distributed CPS translation* ensures control-flow integrity of the contract, which we do on top of a formal definition of the compilation steps. The formally proven secure Prisma compiler eliminates the risk of programmers implementing unsafe interactions that can potentially be exploited.

Contributions. We make the following contributions:

- (1) We introduce Prisma,² a global language for tierless dApps with direct-style client–contract interactions and explicit access control, implemented as an embedded DSL in Scala. Crucially, Prisma automatically enforces the correct program flow (Section 2).
- (2) A core calculus, MiniPrisma, which formalizes both Prisma and its compiler, as well as a proof that our compiler guarantees the preservation of control flow in presence of an attacker that controls the client code (Section 3).
- (3) Case studies which show that Prisma can be used to implement common applications without prohibitive performance overhead (Section 5).
- (4) A comparison of Prisma with a session type and a type state smart contract programming language and the mainstream Solidity/JavaScript programming model (Section 6).

²Prisma implementation and case studies are publicly available: <https://github.com/stg-tud/prisma>

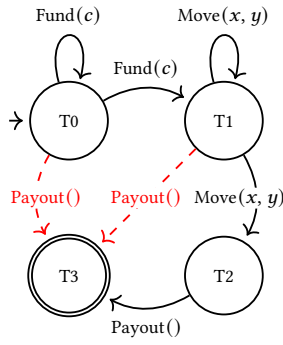


Fig. 1. TicTacToe control flow.

Table 2. Location annotations.

Annotations	Description
<code>@co</code>	on contract
<code>@cl</code>	on clients
<code>@co @cl</code>	independent copies on clients and contract
<code>@co @cross</code>	on contract, but also accessible by client
<code>@cl @cross</code>	(illegal combination)

```

1  @prisma object TicTacToeModule {
2
3  @co @cl case class UU(x: U8, y: U8)
4
5  class TicTacToe(
6    val players: Arr[Address],
7    val fundingGoal: Uint) {
8
9    // u8 is an unsigned 8-bit integer
10   @co @cross var moves: U8 = "0".u8
11   @co @cross var winner: U8 = "0".u8
12   @co @cross val board: Arr[Arr[U8]] =
13     Arr.ofDim("3".u, "3".u)
14
15   @co def performMove(x: U8, y: U8): Unit =
16     { /* ... */ }
17   @cl def updateBoard(): Unit =
18     { /* ... */ }
19   @cl def fund(): (U256, Unit) =
20     (readLine("How much?").u, ())
21   @cl def move(): (U256, UU) =
22     ("0".u, UU(readLine("x-pos?"),
23               readLine("y-pos?")))
24
25   @cl def payout(): (U256, Unit) = {
26     readLine("Press (enter) for payout")
27     ("0".u, ())
28   }
29   @co val init: Unit = {
30     while (balance() < FUNDING_GOAL) {
31       awaitCL(_ => true) { fund() }
32     }
33     while (moves < "9".u && winner == "0".u) {
34       val pair: UU = awaitCL(a =>
35         a == players(moves % "2".u)) { move() }
36       performMove(pair.x, pair.y)
37     }
38     awaitCL(a => true) { payout() }
39     if (winner != "0".u) {
40       players(winner - "1".u).transfer(balance())
41     } else {
42       players("0".u).transfer(balance() / "2".u)
43       players("1".u).transfer(balance()) // remainder
44     }
45   }
46 }

```

Fig. 3. TicTacToe dApp.

2 PRISMA IN A NUTSHELL

We present Prisma by the example of a TicTacToe game, demonstrating that client and contract are written in a single language, where protocols are expressed by control flow (instead of relying on advanced typing disciplines) and enforced by the compiler.

Example. TicTacToe is a two-player game over a 3×3 board. Players take turns in writing their sign into one of the free fields until all fields are occupied, or one player wins by owning three fields in any row, column, or diagonal. The main transaction of a TicTacToe dApp is $Move(x, y)$ used by a player to occupy field (x, y) . A $Move(x, y)$ is valid if it is the sender's turn and (x, y) is empty. Before the game, players deposit their stakes, and after the game, the stakes are paid to the winner.

Fig. 1 depicts possible control flows with transitions labeled by client actions that trigger them. Black arrows depict intended control flows. The dApp starts in the funding state where both parties deposit stakes via $Fund(c)$. Next, parties execute $Move(x, y)$ until one party wins or the game ends

in a draw. Finally, any party can invoke a payout of the stakes via *Payout()*.³ Red dashed arrows illustrate the effects of a mismanaged control flow: a malicious player could trigger a premature payout preventing the counterpart to get financial gains.

Tierless dApps. Prisma is implemented as a DSL embedded into Scala, and Prisma programs are also valid Scala programs.⁴ Prisma interleaves contract and client logic within the same program. Annotations `@co` and `@cl` explicitly place declarations on the contract and on the client, respectively (cf. Tab. 2). A declaration marked as both `@co` and `@cl` has two copies. For security, code placed in one location cannot access definitions from the other – an attempt to do so yields a compile-time error. Developers can overrule this constraint to enable clients to read contract variables or call contract functions by combining `@co` with `@cross`. Combining `@cl` with `@cross` is not allowed – information can only flow from client to contract as part of a client–contract interaction protocol.

There are three kinds of classes. *Located classes* are placed in one location (annotated with either `@co` or `@cl`); they cannot contain located members (annotated with either `@co` or `@cl`) and their instances cannot cross the client–contract boundary, e.g., be passed to or returned from `@cross` functions. *Portable classes* are annotated with both `@co` and `@cl`. Their instances can be passed to and returned from `@cross` functions; they must not contain mutable fields. *Split classes* have no location annotation; their instances live partly in both locations; they cannot be passed to or returned from `@cross` functions and their members must be located.

Prisma code is grouped into modules. While client declarations can use and be used from standard (non-Prisma) Scala code, contract declarations are not accessible from Scala, and can only reference contract code from other Prisma modules (because contract/client code lives in different VMs).

For illustration, consider the TicTacToe dApp (Fig. 3). The `TicTacToeModule` (Line 1) – modules are called **object** in Scala – contains a portable class `UU` (Line 3) and a split class `TicTacToe` (Line 5). Variables `moves`, `winner`, `board` (Lines 10, 11, 13) are placed on the contract and can be read by clients (`@co @cross`). The `updateBoard` function (Line 17) is placed on the client and updates client state (e.g., client’s UI). The `move` function (Line 15) is placed on the contract and changes the game state (`move`). `move` is not annotated with `@cross`, because `@cross` is intended for functions that do not change contract state and can be executed out-of-order without tampering with the client–contract interaction protocol. While Scala only has signed integers and signed longs literals, these are uncommon in Ethereum. Therefore, Prisma provides portable unsigned and signed integers for power-of-two bitsizes between 2^3 to 2^8 , with common arithmetic operations, e.g., `"0".u8` is an unsigned 8-bit integer of value 0 (Line 10).

Encoding client–contract protocols. In Prisma, a client–contract protocol is encoded as a split class containing dedicated `awaitCl` expressions for actively requesting and awaiting messages from specific clients and standard control-flow constructs. Hence, creating a new contract instance corresponds to creating a new instance of a protocol; once created, the contract instance actively triggers interactions with clients. The `awaitCl` expressions have the following syntax:

```
def awaitCl[T](who: Addr => Bool)(body: => (Ether, T)): T
```

They take two arguments. The first (`who`) is a predicate used by clients to decide whether it is their turn and by the contract to decide whether to accept a message from a client. This is unlike Solidity, where a function may be called by any party by default. By forcing developers to explicitly define access control, Prisma reduces the risk of human failure. The second argument (`body`) is the expression to be executed by the client. The client returns a pair of values to the contract: the

³We omit handling timeouts on funding and execution for brevity.

⁴In Scala `val/var` definitions are used for mutable/immutable fields and variables, `def` for methods, `class` for classes, and `object` for singletons. A `case class` is a class whose instances are compared by structure and not by reference.

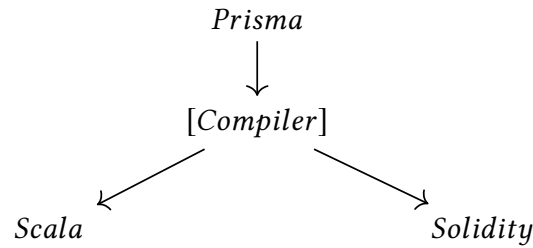
amount of Ether and the message. The former can be accessed by the contract via the built-in expression `value`, the latter is returned by `awaitCl`. Besides receiving funds via `awaitCl`, a contract can also check its current balance (`balance()`), and transfer funds to an account (`account.transfer(amount)`).

Prisma’s programming model is specifically designed to accommodate blockchain use cases. In contrast to other tierless models like client-server, we emphasise inversion of control such that the code is written as if the contract was the active driver of the protocol, while clients are passive and only react to requests by the contract. This enables to enforce the protocol on the contract side. For this reason, for example, we support the `awaitCl` construct on the active contract side whereas there is no corresponding construct on the passive client side.

For illustration, consider the definition of `init` on the right-hand side of Fig. 3, Line 28. It defines the protocol of `TicTacToe` as follows. From the beginning of `init`, the flow reaches `awaitCl` in Line 30 where the contract waits for clients to provide funding (by calling `fund`). Next, the contract continues until `awaitCl` in Line 33 and clients execute `move` (Line 21) until the game ends with a winner (`winner != 0`) or a draw (`moves >= 9`). At this point – `awaitCl` in Line 37 – any party can request a payout and the contract executes to the end. The example illustrates how direct-style `awaitCl` expressions and the tierless model enable encoding multiparty protocols as standard control flow, with protocol phases corresponding to basic blocks between `awaitCl` expressions.

Compiling Prisma to Solidity. Abstractly, Prisma’s compiler takes a Prisma dApp program and splits it into two separate programs: A Scala client program and a Solidity contract program (Fig. 4a). In more detail, the compiler (1) places all definitions according to their annotations and (2) splits contract methods that contain `awaitCl` expressions into a method that contains the code up to the `awaitCl` and a method that contains the continuation after the `awaitCl` (taking the result of the `awaitCl` as an argument). Once deployed, a contract is public and can be messaged by arbitrary clients – not exclusively the ones generated by Prisma – hence, we cannot assume that clients will actually execute the body passed to them by an `awaitCl` expression. To cope with malicious clients trying to tamper with the control flow of the contract, the compiler hardens contract code by generating code to enforce *control flow integrity*: storing the current phase before giving control to the client and rejecting methods invoked by wrong clients or in the wrong phase.

For illustration, the code generated from Fig. 3 is schematically shown in Fig. 4b and 4c. The methods `updateBoard`, `fund`, `move`, and `payout` are annotated `@cl` and thus compiled into the client program (Fig. 4b). The variables `moves`, `winner` and `board`, and the method `performMove` are annotated `@co` and thus compiled into the contract program (Fig. 4c). Further, three new methods are generated on both the client and the contract – one for each `awaitCl` expression in `init` – corresponding to phases in the logical protocol (Fig. 1). The `Funding` method of the client (Line 16) is generated from the body of the first `awaitCl`. Similarly, the `Move` method (Line 18) is generated from the second `awaitCl` and the `Payout` method (Line 20) from the third `awaitCl`. In the example, the generated methods are given meaningful names by capitalizing the single method called in the body of the `awaitCl` expressions form which they were generated. In the actual implementation, generated methods are simply enumerated. The code up to the first `awaitCl` (Line 30, Fig. 3) is placed in the constructor of the generated contract, which ends by setting the active phase to `Funding`. The code between the first and the second `awaitCl` either loops back to the first `awaitCl` or continues to the second one (Line 33). The code is placed in the `Fund` method that requires the phase to be `Funding`, and may change it to `Exec` if the loop condition fails. Similarly, the method `Move` is generated to contain the loop between the second and the third `awaitCl` (Line 37); and the method `Payout` contains the code from the third `awaitCl` to the end of `init`. Only the second `awaitCl` contains a (non-trivial) access control predicate, which results in an additional assertion in the body of `Move` (Line 46, Fig. 4a). Observe that the



(a) Compilation scheme

```

1  class TTT {
2
3      // @cl annotated definitions
4      def updateBoard(): Unit =
5          { /* ... */ }
6      def fund(): (U256, Unit) =
7          (readLine("How much?").u, ())
8      def move(): (U256, UU) =
9          ("0".u, UU(readLine("x-pos?"),
10             readLine("y-pos?")))
11     def payout(): (Ether, Unit) = {
12         readLine("Press (enter) for payout")
13         ("0".u, ()) }
14
15     // body of awaitCl expressions
16     def Fund(): (Ether, Unit) =
17         fund()
18     def Move(): (Ether, UU) =
19         move()
20     def Payout(): (Ether, Unit) =
21         payout()
22
23     /* ... */
24
25
26
27
28 }
  
```

(b) Scala client

```

29  contract TTT {
30      State phase = T0; enum State {T0, T1, T2, T3}
31
32      // @co annotated definitions
33      int moves = 0;
34      int winner = 0;
35      int[][] board;
36      function performMove(int x, int y) private { /*...*/ }
37
38      // continuation of awaitCl expressions
39      function Fund() public {
40          require(phase == T0);
41          /*...*/;
42          if (!(balance < FUNDING_GOAL)) phase = T1;
43          /* else phase remains T0; this models the first while loop */
44      }
45      function Move(int x, int y) public {
46          require(phase == T1 && sender == players(moves % 2));
47          /*...*/;
48          if (!(moves < 9 && winner == 0)) phase = T2;
49          /* else phase remains T2; this models the second while loop */
50      }
51      function Payout() public {
52          require(phase == T2);
53          /*...*/;
54          phase = T3;
55      }
56 }
  
```

(c) Solidity contract

Fig. 4. TicTacToe dApp after compilation, simplified

return types of the generated client methods are the argument types of the corresponding contract methods.

Compilation Techniques. While CPS is a key step in our translation pipeline, the example shows the final defunctionalised, trampolined code. The final output does not contain explicit continuations (i.e., a function that takes another function as an argument and calls that as its continuation). Instead, after defunctionalizing and trampolining the CPS translation, only one top-level function (Fund, Move, Payout) is callable at each phase, which is ensured by the `require` statement at the beginning of each function, and each function sets the next phase at the end. These functions play the role of the continuations.

Let us look at the correspondence between the original Prisma code (Fig. 3) and the generated Solidity code (Fig. 4c) from a higher-level perspective: To verify that the Prisma code matches the generated Solidity code, we proceed as follows.

First, we verify that the control flow of Fig. 3 is accurately described by the automaton diagram in Fig. 1. In particular, we observe that there are two loops in the automaton and there are also two `while` loops in the Prisma code. Further, there are three `awaitCl` expressions in the code, and there are three states in the automaton (plus a final state).

Second, we verify that the automaton in Fig. 3 corresponds to the program flow of the Solidity code in Fig. 4c. In particular, we observe that there are four states in the automaton and there are four states in the Solidity code. Three of those have an associated function (τ_0 is `Fund`, τ_1 is `Move`, τ_2 is `Payout`), which are the only public functions that can be invoked in that state, thanks to the `require` statements. In the final state τ_3 , no public function can be invoked. Furthermore, we can see that the automaton has two loops. It is possible to go from τ_0 either to τ_1 or stay in τ_0 . This is represented in the Solidity code, by checking for the loop condition at the end of the function associated to τ_0 , and then either changing the phase to τ_1 , or doing nothing, which means staying in τ_0 . Similarly, the loop in state τ_1 is encoded with an `if` at the end of the function to conditionally move to the next phase.

These two steps should illustrate how the control flow of the Prisma program – which is abstractly visualized by the automaton – is implemented and enforced by the generated Solidity program.

3 COMPILATION AND ITS CORRECTNESS

We informally introduce Prisma’s compilation process and our notion of correctness before formally specifying and proving the compiler correct.

3.1 High-level Overview of Prisma’s Secure Compilation

To implement the contract-client interaction, we CPS-translate Prisma code and execute continuations alternately between contract and client. A standard CPS translation is, however, not sufficient because the control flow is distributed and we need to send function calls (i.e., the current continuation) over the network – or, more specifically, send the name and the arguments of the next function to execute. For this, we defunctionalise [71] the code to turn functions calls (which represent continuations) into data. This compilation process performs an *inversion of control* between the contract and the client. With Prisma’s contract–client communication in direct style, we can write dApps as if *the contract* was in control of the execution; Prisma allows the contract to request messages from clients and to process only responses that it requested.

After the compilation process, clients are in control of the execution because, in blockchains, contracts purely respond to messages from clients. As a result, dApps may become the target of malicious attacks. In our security model, we trust the contract to execute the code that we generate for it, whereas we consider the client code untrusted, i.e., the client side can run arbitrary code. Crucially, it could pass unintended continuations to the contract to force the execution to continue in an arbitrary state. For example, in the source code of the TicTacToe game (Section 2), one needs to go through the game loop after funding and before payout. Yet, the compiled code is separated and distributed into small chunks. Parties execute a chunk and then wait for other parties to decide on a move that influences how to proceed with the execution. For this reason, the client could send a message at any time telling the contract to go into the payout phase. We need to guard the contract against such attempts to make it deviate from the protocol. Conceptually, if the client was able to force the execution to continue in an arbitrary state, the control flow in the Prisma source would be violated. Execution would ‘jump’ from one client expression to another one skipping the code in between, which is not possible with the semantics of the source language.

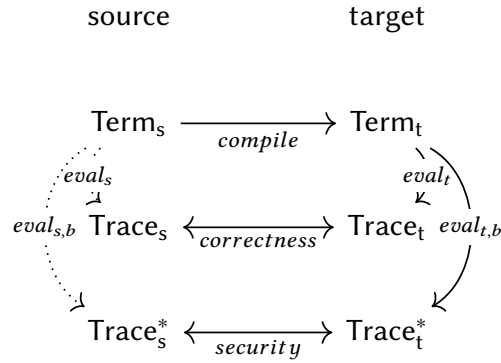


Fig. 5. Secure Compilation

Prisma’s compiler avoids such attacks and preserves control flow by inserting guards on the contract side. Guards are in places where the basic blocks of the program have been separated and distributed onto different hosts by CPS translation – to reject any improper continuations from clients. Guarding ensures the control flow integrity [1] of the contract in the presence of malicious clients by excluding any behavior of the compilation target that cannot be observed from the source. Informally, this is our notion of *secure compilation*, which we rigorously define and prove for Prisma’s compiler in this section. The compilation process is key in hiding the complexity of enforcing distributed control flow from the developer – hence, a formal proof of its correctness is critical.

To formalize the compiler, we specify a source and a target language. Fig. 5 shows a schema of our compilation and the proof. The compiler (Fig. 5, top) is a function that maps terms in the source language ($Term_s$) into terms in the target language ($Term_t$). A correct compiler preserves some properties of the code – depending on the notion of correctness. For example, typeability-preserving and semantics-preserving compilers have been extensively studied [63]. Because types are not the focus of this paper, we omitted them from the figure. In the middle part of Fig. 5, we show the evaluation of source and target to traces ($eval_s$ and $eval_t$, respectively) – and traditional compiler correctness as the equivalence between traces generated from the sources ($Trace_s$) and from the target ($Trace_t$). But compiler correctness⁵ in this traditional sense is not sufficient in the presence of malicious attackers that can tamper with parts of the code. Instead, we need to prove that Prisma is a *secure abstraction*, i.e., if security problems can arise on the target, they must be visible in the Prisma source code, too, so that developers do not need to look at target code to reason about potentially misbehaving clients. To this end, we define a hypothetical *attacker model on the source code* as the ability to only replace the body of a Prisma client expression and show that, with the contract part hardened with guards, the target attacker does not gain additional power over the hypothetical source attacker. Specifically, we define malicious semantics $eval_{t,b}$ and $eval_{s,b}$ for the target and the source language, respectively, and show that $eval_{t,b}(compile\ e) = compile(eval_{s,b}\ e)$ (security property in Fig. 5).

In the remainder of this section:

- We present the core calculus (Section 3.2) $MiniPrisma_*$ – a hybrid language that includes elements of both the source ($MiniPrisma_s$) and the compilation target ($MiniPrisma_t$), while abstracting over details of both Scala and Solidity. We define a hybrid language because the source and the target share many constructs – the hybrid language allows us to focus on how the differences are compiled.

⁵Type and semantics preservation is not the focus of this paper; we presume them for our compiler without a formal proof.

	$id \in ID$	$i \in I$	$j \in \{\text{who, state, clfn, cofn}\}$
(definition)		$d ::= @co \text{ this.}i = v; d \mid @cl \text{ this.}i = v; d \mid ()$	
(synthetic definition)		$b ::= @co \text{ this.}j = v; b \mid @cl \text{ this.}j = v; b \mid ()$	
(program)		$P ::= d; b; m$	
(constant)		$c ::= 0 \mid 1 \mid 2 \mid \dots \mid \text{true} \mid \text{false} \mid () \mid \&\& \mid + \mid == \mid < \mid \text{try}$ $\mid \gg= \mid \text{trmp} \mid \text{Done} \mid \text{More}$	
(value)		$v ::= c \mid v :: v \mid x \rightarrow e$	
(pattern)		$x ::= c \mid x :: x \mid id$	
(expression)		$e ::= c \mid e :: e \mid x \rightarrow e \mid id \mid x=e; e \mid e e$ $\mid \text{this.}i \mid \text{this.}i := e \mid \text{this.}j \mid \text{this.}j := e$	
(main expression)		$m ::= c \mid m :: m \mid x \rightarrow e \mid id \mid x=m; m \mid m m$ $\mid \text{this.}i \mid \text{this.}i := m \mid \text{this.}j \mid \text{this.}j := m$ $\mid \text{awaitCl}_s(e, () \rightarrow e) \mid \text{awaitCl}_t(c, () \rightarrow e)$	

Fig. 6. MiniPrisma_{*} syntax.

- We model the compiler (Section 3.3) as a sequence of steps that transform MiniPrisma_{*} programs via several intermediate representations.
- We define MiniPrisma_{*} semantics as a reduction relation over configurations consisting of traces of evaluation events and expressions being evaluated (Section 3.4). We distinguish between a good semantics, which evaluates the program in the usual way, and a bad semantics, which models attackers by ignoring client instructions and producing arbitrary values that are sent to the contract.
- We prove secure compilation by showing that the observable behavior of the programs before and after compilation is equivalent (Section 3.5). We capture the observable program behavior by the trace of events generated during program evaluation (as guided by the semantic definition) and show trace equivalence of programs before and after compilation.

3.2 Syntax

The syntax of MiniPrisma_{*} (Fig. 6), has three kinds of identifiers id , i , j , from unspecified sets of distinct names. Pure identifiers id are for function arguments and let bindings; mutable variables i are for heap variable assignment and access. In the target program, mutable variables j (who, state, clfn, cofn) generated by the compiler can also appear. We call compiler-generated identifiers *synthetic*. Normal identifiers are separated from synthetic ones to distinguish compiler generated and developer code. Definitions d and definitions for synthetic identifiers b are semicolon-separated lists of declarations that assign values to variables and annotate either the contract or the client location. Each program P consists of definitions d and synthetic definitions b followed by the main contract expression m . Program P corresponds to a single Prisma split class, d and b to methods and generated methods, and m to a constructor containing the initialisation of its class members (such as the body of `init`, Fig. 3).

Constants c are unsigned 256 bit integer literals and built-in operators. MiniPrisma_{*} supports tuples introduced by nesting pairs ($::$) and eliminated by pattern matching. Tuples allow multiple values to cross tiers in a single message. Values v are constants, value pairs, and lambdas. Patterns x are constants, pattern pairs, and variables. Expressions e are constants, expression pairs, lambdas, variables, variable accesses/assignments, bindings and function applications.

$$\begin{aligned}
m_0 \ c \ m_1 &= c(m_0, m_1) \\
(m_0, \dots, m_n) &= m_0 :: \dots :: m_n :: () \\
m_0; m_1 &= () = m_0; m_1 \\
\text{assert}(m_0); m_1 &= \text{true} = m_0; m_1 \\
x \leftarrow e_1; m_2 &= x = \text{awaitCl}_t(() \rightarrow e_1); m_2 \\
\text{if let } x = m_1 \text{ then } e_2 \text{ else } e_3 &= \text{try}(m_1, x \rightarrow e_2, () \rightarrow e_3)
\end{aligned}$$

Fig. 7. Syntactic sugar.

Main expressions m may further contain remote client expressions, embedding client code into contract code and waiting for its result. The source client expression $\text{awaitCl}_s(e, () \rightarrow e)$ can be answered by any client whose address fulfills the predicate specified as first argument. awaitCl_s corresponds to direct-style remote access via `awaitCl` in Prisma. We use the syntax form $\text{awaitCl}_t(c, () \rightarrow e)$ to model the execution of code e on the specified client c . awaitCl_t has no correspondence in the source syntax. Our compilation first splits the predicate from the source client expressions into a separate access control guard. Then, it eliminates client expressions, turning the contract into a passive entity that stops and waits for client input.

We now map the hybrid language MiniPrisma_* to the source and target languages, MiniPrisma_s and MiniPrisma_t . MiniPrisma_s has all expressions of MiniPrisma_* , except those that contain $\gg=$ (bind), `trmp` (trampoline), `Done`, `More`, awaitCl_t , or synthetic identifiers j . MiniPrisma_t has all expressions of MiniPrisma_* except those that contain awaitCl_s , awaitCl_t , $\gg=$.

$\gg=$ and awaitCl_t may not appear neither in source nor target programs; the former is used only as an intermediate construction for the compiler, the latter only during evaluation to track the current location.

Syntactic sugar. In Fig. 7, we define some syntactic sugar to improve readability. We use infix binary operators and tuple syntax for nested pairs ending in the unit value $()$; we elide the `let` expression head for `let` bindings matching $()$, $\text{assert}(x)$ is a `let` binding matching `true`; we use monadic syntax for `let` bindings of effectful expressions; `if let $x = m$ then e_2 else e_3` is the application of the built-in `try` function.

Events and configurations. In Fig. 8, we define left-to-right evaluation contexts E [34]; and compilation frames F [66], such that every expression decomposes into a frame-redex pair $F e$ or is an atom a . Events p and q are lists that capture the observable side-effects of evaluating expressions. They are either (a) state changes $wr(c, i, v)$ and $wr(c, j, v)$, from the initial definitions or variable assignment, where i and j are the variable being assigned, c the location, and v the assigned value, or (b) client-to-contract communication $\text{msg}(c, v)$, where c is the address of the client and v the sent value. Configurations $C = p; q; cm$, represent a particular execution state, where p (and q) are traces of normal (and synthetic) events produced by the evaluation, c is the evaluating location, and m is the expression under evaluation.

Initialization. Initialization in Fig. 9 generates the initial program configuration, which models the decentralized application with a single contract and multiple clients. We model a fixed set of clients A interacting with a contract. The initialization of a program $d; b; m$ to a configuration $p; q; 0; m$ leaves the expression m untouched and generates a list of events – one write event for each normal and synthetic definition. Location 0 represents the contract.

(frame)	$F ::= \text{awaitCl}_s(\square, () \rightarrow e) \mid \square e \mid e \square \mid \square :: e \mid e :: \square$ $\mid x = \square; e \mid x = e; \square \mid \text{this}.i := \square \mid \text{this}.j := \square$
(atom)	$a ::= \text{this}.i \mid \text{this}.j \mid c \mid id \mid x \rightarrow e$
(context)	$E ::= \square \mid E :: m \mid v :: E \mid E m \mid v E \mid x = E; m \mid \text{this}.i := E \mid \text{this}.j := E$
(event)	$p ::= \text{wr}(c, i, v) p \mid \text{msg}(c, v) p \mid ()$
(synthetic event)	$q ::= \text{wr}(c, j, v) q \mid ()$
(configuration)	$C ::= p; q; c; m$

Fig. 8. Frames, Events and configurations.

$$\begin{aligned}
init_A(d; b; m) &= init_A(d; b); 0; m \\
init_A(d; b) &= (\text{wr}(0, i, v) \mid \forall (@\text{co this}.i = v) \in d) \\
&\quad (\text{wr}(0, j, v) \mid \forall (@\text{co this}.j = v) \in b) \\
&\quad (\text{wr}(c, i, v) \mid \forall (@\text{cl this}.i = v) \in d, c \in A) \\
&\quad (\text{wr}(c, j, v) \mid \forall (@\text{cl this}.j = v) \in b, c \in A)
\end{aligned}$$

Fig. 9. Initialization.

3.3 Compilation

The compiler eliminates language features not supported by the compilation target one by one, lowering the abstraction level from (1) *direct style communication (DS)* – which needs language support for !-notation [10] – through the intermediate representations of (2) *monadic normal form (MNF)* – which needs support for do-notation [53] – and (3) *continuation-passing style (CPS)* – which needs higher-order functions – to (4) explicitly encoding *finite state machines (FSM)* – for which first-order functions suffice. In the following, we provide an intuition for the compiler steps and subsequently their formal definitions.

First, the compilation steps *mnf* and *assoc* transform DS remote communication $\text{awaitCl}_s(e, () \rightarrow e)$ to variable bindings ($id := e$) and nested let bindings are flattened such that a program is prefixed by a sequence of let expressions. Second, step *guard* generates access control guards around client expressions to enforce correct execution even when clients behave maliciously. Third, step *cps* transforms previously generated let bindings for remote communication ($x \leftarrow e_1; m_2$) to monadic bindings $e_1 \gg x \rightarrow m_2$. Fourth, step *defun* transforms functions into data structures that can be sent over the network and are interpreted by a function (i.e., an FSM) on the other side. Compared to standard defunctionalization, we handle two more issues. First, we defunctionalize the built-in higher-order operator (\gg) by wrapping the program expression into a call to a trampoline $\text{trmp}(\dots)$ and transforming the bind operator ($\dots \gg x \rightarrow \dots$) to the (More, ..., ...) data structure; the trampoline repeatedly interprets the argument of More until it returns Done instead of More signaling the program’s result. Second, we keep contract and client functions separate by generating separate synthesized interpreter functions, called *cofn* and *clfn*, thereby splitting the code into the parts specific to contract and client.

MNF transformation (Fig. 10). The *mnf'* function wraps the main expression m into a call to the trampoline with the pair (Done, m) – signaling the final result – as argument. Then, *mnf* transforms expressions recursively, binding sub-expressions to variables, resulting in a program prefixed by a sequence of let bindings. As recursive calls to *mnf* may return chains of let bindings, we apply *assoc* to produce a flat chain of let bindings. Given a let binding, whose sub-expressions are in MNF, associativity recursively flattens the expression, by moving nested let bindings to the

$$\begin{aligned}
mnf'(d; b; m) &= d; b; \text{trmp}(mnf((\text{Done}, m))) \\
mnf(F e) &= \text{assoc}(id_0=e; mnf(F id_0)) \\
mnf(a) &= a \\
\text{assoc}(x_0=(x_1=m_1; m_0); m_2) &= \text{assoc}(x_1=m_1; \text{assoc}(x_0=m_0; m_2)) \\
\text{assoc}(m) &= m
\end{aligned}$$

Fig. 10. Monadic normal form transformation.

$$\begin{aligned}
\text{guard}'(d; b; \text{trmp}(m)) &= d; b; \text{trmp}(\text{guard}(m)) \\
\text{guard}\left(\begin{array}{l} x \leftarrow_s (e_0, () \rightarrow e_1); \\ m_2 \end{array}\right) &= \left(\begin{array}{l} \text{this.who} := e_0; \text{this.state} := c; \\ x \leftarrow_s (() \rightarrow \text{true}, () \rightarrow e_1); \\ \text{assert}(\text{this.state} == c \ \&\& \\ \quad \text{this.who}(\text{this.sender})); \\ \text{this.state} := 0; \text{guard}(m_2) \end{array} \right) \\
&\quad \text{where } c \text{ fresh} \\
\text{guard}(x = e_0; m_1) &= x = e_0; \text{guard}(m_1) \\
\text{guard}(m) &= m
\end{aligned}$$

Fig. 11. Guarding.

$$\begin{aligned}
\text{cps}'(d; b; \text{trmp}(m)) &= d; b; \text{trmp}(\text{cps}(m)) \\
\text{cps}(x \leftarrow_s (() \rightarrow \text{true}, e_0); m_1) &= e_0 \gg= (x \rightarrow \text{cps}(m_1)) \\
\text{cps}(x = e_0; m_1) &= x = e_0; \text{cps}(m_1) \\
\text{cps}(m) &= m
\end{aligned}$$

Fig. 12. Continuation-passing style transformation.

front, (... (... $m_0; m_1$); $m_2 = \dots m_0; (\dots m_0; m_2)$), creating a single MNF expression (i.e., *assoc* is composition for MNF terms).

Guarding (Fig. 11). We insert access control guards for remote communication expressions \leftarrow_s to enforce (i) the execution order of contract code after running the client expression and (ii) that the correct client invokes the contract continuation. The transformation sets the synthetic variable *state* to a unique value before the client expression, and stores the predicate to designate valid clients in the synthetic variable *who*. After the client expression, the generated code asserts that the contract is in the same state, and checks that the sender fulfills the predicate. The assertion trivially holds in the sequential execution of the source language, but after more compilation steps the client will be responsible for calling the correct continuation on the contract. Since client code is untrusted, the contract needs to ensure that only the correct client can invoke only the correct continuation.

CPS transformation (Fig. 12). The *cps* transformation turns the chains of let bindings produced by *mnf* into CPS. The chain contains three cases of syntax forms: (1) monadic binding ($x \leftarrow \dots; m_1$), (2) let binding ($x = e_0; m_1$), or (3) final expression. For (1), *cps* replaces the monadic binding with an explicit call to the bind operator (... $\gg= (x \rightarrow \text{cps}(m_1))$). For (2) and (3), *cps* recurses into the tail of the chain. This resembles do-notation desugaring (e.g., in Haskell).

$$\begin{aligned}
\text{defun}'(d; b; e) &= \text{defun}(d; \text{coclfn}(b, id, \text{assert}(\text{false}), \text{assert}(\text{false})); e) \\
&\quad \text{where } id \text{ fresh} \\
\text{defun} \left(\begin{array}{l} d; \text{coclfn}(b, id, \\ e_{1,alt}, \\ e_{2,alt} \\); \\ ((\rightarrow e_1) \gg= (x \rightarrow e_2)) \end{array} \right) &= \left(\begin{array}{l} d; \text{coclfn}(b, id, \\ \text{if let } (c :: fv(\rightarrow e_1)) = id \\ \text{then } e_1 \text{ else } e'_{1,alt}, \\ \text{if let } (c :: x :: fv(x \rightarrow e'_2)) = id \\ \text{then } e'_2 \text{ else } e'_{2,alt}); \\ (\text{More}, c :: fv(\rightarrow e_1), c :: fv(x \rightarrow e'_2)) \end{array} \right) \\
&\quad \text{where } c \text{ fresh} \\
&\quad \text{and } d; \text{coclfn}(b, id, e'_{1,alt}, e'_{2,alt}); e'_2 = \\
&\quad \quad \text{defun}(d; \text{coclfn}(b, id, e_{1,alt}, e_{2,alt}); e_2) \\
\text{defun} \left(\begin{array}{l} d; \text{coclfn}(b, id, \\ e_{1,alt}, e_{2,alt}); \\ x = e_0; e_1 \end{array} \right) &= d; \text{coclfn}(b, id, e_{1,alt}, e_{2,alt}); x = e_0; \text{defun}(e_1) \\
\text{defun} \left(\begin{array}{l} d; \text{coclfn}(b, id, \\ e_{1,alt}, e_{2,alt}); \\ e \end{array} \right) &= d; \text{coclfn}(b, id, e_{1,alt}, e_{2,alt}); e \\
\text{coclfn}(b, id, e_{1,alt}, e_{2,alt}) &= @cl \text{ this. clfn} = id \rightarrow e_{1,alt}; \\
&\quad @co \text{ this. cofn} = id \rightarrow e_{2,alt}; b
\end{aligned}$$

Fig. 13. Defunctionalization.

Defunctionalization (Fig. 13). The *defun* function transforms the chains of let bindings and bind operators produced by *cps*, which contains three cases of syntax forms: (1) a bind operator ($e_1 \gg= e_2$), or (2) a let binding ($x = e_1; e_2$), or (3) the final expression. For (1), e_1 and e_2 are replaced by data structures that contain values for the free variables in e_1 and e_2 and are tagged with a fresh ID. The body of the expression is lifted to top-level synthetic definitions. For this, *defun* modifies the synthetic definitions b by extracting the body $e_{1,alt}$ of the synthetic *clfn* definition and the body $e_{2,alt}$ of *cofn*, and by adding an additional conditional clause to these definitions. The added clause answers to requests for a given ID with evaluating the original expression. For (2) and (3), *defun* recurses into the expressions.

After defunctionalization, lambdas $x \rightarrow e_0$ are lifted and assigned a top-level identifier id_0 and lambda applications, $id_0(e_1)$, are replaced with calls to a synthesized interpreter function $\text{fn}(id_0, e_1)$. The latter branches on the identifier and executes the code that was lifted out of the original function.

Compiling. The *comp* function composes the compiler steps (not including *mnf*). We also define the *comp'* function, which jumps over the wrapping *trmp* expression and initialises the defunctionalisation with an environment that contains the two functions *cofn* and *clfn*, which assert false.

$$\begin{aligned}
\text{comp} &= \text{defun} \circ \text{cps} \circ \text{guard} \\
\text{comp}' &= \text{defun}' \circ \text{cps}' \circ \text{guard}'
\end{aligned}$$

3.4 Semantics

We model the semantics as a reduction relation over configurations $p; q; c; m \rightarrow p'; q'; c'; m'$. Location $c = 0$ denotes contract execution, otherwise execution of client of address c . We distinguish

$$\begin{array}{l}
\text{(Rgs)} \quad p; q; 0; \text{awaitCl}_s(v, () \rightarrow e) \quad \rightarrow_g \quad p; q; 0; \text{awaitCl}_t(c, () \rightarrow e) \quad \text{if } p; q; 0; v(c) \rightarrow^* p; q; 0; \text{true} \\
\text{(Rbs)} \quad p; q; 0; \text{awaitCl}_s(v, () \rightarrow e) \quad \rightarrow_b \quad p; q; 0; \text{awaitCl}_t(c, () \rightarrow e) \\
\text{(RTM)} \quad p; q; 0; \text{trmp} \left(\begin{array}{l} \text{More,} \\ v_1 :: e_1, \\ v_2 :: e_2 \end{array} \right) \quad \rightarrow \quad p; q; 0; \left(\begin{array}{l} id = \text{awaitCl}_t(c, \text{this.cfn}(v_1 :: e_1)); \\ \text{trmp}(\text{this.cofn}(v_2 :: e_2)) \end{array} \right) \\
\text{(RTD)} \quad p; q; 0; \text{trmp}(\text{Done}, v) \quad \rightarrow \quad p; q; 0; v \\
\text{(RG)} \quad p; q; 0; \text{awaitCl}_t(c, () \rightarrow e) \quad \rightarrow_g \quad p; q \text{ msg}(c, v) \text{ wr}(0, \text{sender}, c); 0; v \quad \text{if } p; q; c; e \rightarrow^* p'; q'; c; v \\
\text{(RB)} \quad p; q; 0; \text{awaitCl}_t(c, () \rightarrow e) \quad \rightarrow_b \quad p; q \text{ msg}(c, v') \text{ wr}(0, \text{sender}, c); 0; v'
\end{array}$$

Fig. 14. Evaluation (1/2).

good (\rightarrow_g) and bad (\rightarrow_b) evaluations (Fig. 14 and 15); shared rules are in black, without subscript (\rightarrow).

Attacker model. Attackers can control an arbitrary number of clients and make them send arbitrary messages. Hence, the bad semantics can answer a request to a client with an arbitrary message from an arbitrary id . We use evaluation with bad semantics to show that our compiler enforces access control against malicious clients.

Good evaluations of client expressions in the source language (Rgs) reduce to a client expression with a fixed client that fulfils the given predicate. We require that predicates evaluate purely. Hence, p and q do not change in the evaluation. On the other hand, bad evaluation of client expressions in the source language (Rbs) ignores the predicate, choosing an arbitrary client. Similarly, bad evaluation also chooses an arbitrary client for the evaluation of a trampoline in the target (RTM), which does not specify a predicate. The trampoline ends when it reaches Done (RTD). Further, after choosing a client to evaluate, the good evaluation (RG) continues to reduce the client expression to a value, while the bad evaluation (RB) replaces the expression e with a (manipulated) arbitrary value v' . Both evaluations (RG, RB) emit the message event $\text{msg}(c, v)$ and an assignment to the special variable sender , when a client expressions is reduced to a value v , to record the client–contract interaction.

Common Evaluation (Fig. 15). Expressions are reduced under the evaluation context E on the current location (RE), assignment to variables is recorded in the trace (RSET $^\circ$), accessing a variable is answered by the most recent assignment to it from the trace in the current location (RGET $^\circ$). For synthetic variables, we use the synthetic store (RGET † , RSET †). Binary operators are defined as unsigned 256 bit integer arithmetic; we only show the rule for addition (ROP). Further, we give rules for conditionals (RT, RF), let binding (RLET) and function application (RLAM) using pattern matching.

Pattern matching (Fig. 16). Matching $[x \Rightarrow v]$ is a partial function, matching patterns x with values v , returning substitution of variables id to values. Matching is recursively defined over pairs; it matches constants to constants, identifiers to values by generating substitutions, and fails otherwise. Substitutions $[id \mapsto v]$, in turn, can be applied to terms e , written $[id \mapsto v] e$ (capture-avoiding substitution). Substitutions σ compose right-to-left $(\sigma\sigma')x = \sigma(\sigma'x)$.

3.5 Secure Compilation

We prove that the observable behavior of the contract before and after compilation is equivalent. We capture the observable behavior by execution traces and show that trace equivalence holds even when the program is attacked, i.e., reduced by \rightarrow_b^* .

Modelling Observable Behavior. The only source of observable nondeterminism in the bad semantics is the evaluation of awaitCl_s and awaitCl_t . As clients decisions on message sending are influenced by the state of contract variables, tracking incoming client messages and state changes in

(RE)	$p; q; 0; E[m]$	$\rightarrow p'; q'; 0; E[m']$	if $p; q; 0; m \rightarrow p'; q'; 0; m'$
(RGET ^o)	$p; q; c; \text{this}.i$	$\rightarrow p; q; c; v$	if $\text{wr}(c, i, v) \in p$
(RGET [†])	$p; q; c; \text{this}.j$	$\rightarrow p; q; c; v$	if $\text{wr}(c, j, v) \in q$
(RSET ^o)	$p; q; c; \text{this}.i := v$	$\rightarrow p \text{ wr}(c, i, v); q; c; ()$	
(RSET [†])	$p; q; c; \text{this}.j := v$	$\rightarrow p; q \text{ wr}(c, j, v); c; ()$	
(ROP)	$p; q; c; v_0 + v_1$	$\rightarrow p; q; c; v'$	if $v' = v_0 + v_1$
(RT)	$p; q; c; \left(\begin{array}{l} \text{if let } x = v \\ \text{then } e_0 \text{ else } e_1 \end{array} \right)$	$\rightarrow p; q; c; e'_0$	if $e'_0 = [x \Rightarrow v] e_0$
(RF)	$p; q; c; \left(\begin{array}{l} \text{if let } x = v \\ \text{then } e_0 \text{ else } e_1 \end{array} \right)$	$\rightarrow p; q; c; e_1$	otherwise
(RAPP)	$p; q; c; (x \rightarrow e) v$	$\rightarrow p; q; c; e'$	if $e' = [x \Rightarrow v] e$
(RLET)	$p; q; c; (x = v; m)$	$\rightarrow p; q; c; m'$	if $m' = [x \Rightarrow v] m$

Fig. 15. Evaluation (2/2).

$$\begin{aligned}
[c \Rightarrow c] &= [] \\
[id \Rightarrow v] &= [id \mapsto v] \\
[(e_0 :: e_1) \Rightarrow (e'_0 :: e'_1)] &= [e_0 \Rightarrow e'_0] \cdot [e_1 \Rightarrow e'_1]
\end{aligned}$$

Fig. 16. Pattern matching.

the trace suffices to capture the observable program behavior. If the observable behavior is the same for the source and the compiled programs, they are indistinguishable. Thus, behavior preservation amounts to trace equality on programs before and after compilation. Further, it suffices to model equality for non-stuck traces. The evaluation gets stuck (program crash) on assertions that guard against deviations from the intended program flow. The Ethereum Virtual Machine reverts contract calls that crash, i.e., state changes of crashed calls do not take effect, hence, stuck traces are not observable.

Since bad evaluation is nondeterministic, we work with not just programs, expressions and configurations, but program sets, expression sets, and configuration sets. Let $p; q; m \Downarrow$ be the trace set of the configuration $p; q; 0; m$, e.g., the set of tuples of the final event sequence p' and value v of all reduction chains that start in $p; q; 0; m$ and end in $p'; 0; q'; v$. Our trace set definition does not include synthetic events q' of the final configuration. Synthetic events are introduced through compilation; excluding them allows us to put source and target trace sets in relation. Further, let the trace set of a configuration set $T \Downarrow$, be the union of the trace sets for each element:

$$\begin{aligned}
p; q; m \Downarrow &= \{ (p', v) \mid (p; q; 0; m) \xrightarrow{*}_b (p'; q'; 0; v) \} \\
T \Downarrow &= \bigcup_{p; q; m \in T} p; q; m \Downarrow
\end{aligned}$$

We say that two configuration sets T and S are equivalent, denoted by $T \approx S$, iff T and S have the same traces sets:

$$(T \approx S) \Leftrightarrow (T \Downarrow = S \Downarrow)$$

By this definition, two expressions that eventually evaluate to the same value with the same trace are related by trace equality. We use this notion of trace equality to prove that a source program is trace-equal to its compiled version by evaluating the compiled program forward $\xrightarrow{*}_b$ and the original program backward $\xleftarrow{*}_b$ until configurations converge.

Secure Compilation. Theorem 1 states our correctness property, which says that observable traces generated by the malicious evaluation of programs are preserved (\approx) by compilation. The malicious evaluation models that client code has been replaced with arbitrary code, while contract code is unchanged. The preservation of observable traces implies the integrity of the (unchanged) contract code. Secure compilation guarantees that developers can write safe programs in the source language without knowledge about the compilation or the distributed execution of client/contract tiers.

THEOREM 1 (SECURE COMPILATION). For each program P over closed terms, the trace set of the program under attack equals the trace set of the compiled program under attack:

$$\forall P. \{ \text{init}_A(\text{comp}'(\text{mnf}'((P)))) \} \approx \{ \text{init}_A(P) \}.$$

We first show that trace equality holds for the different compiler steps. Some compiler steps are defined as a recursive term-to-term transformation on open terms, whereas traceset equality is defined by reducing terms to values, i.e., on closed terms. Since all evaluable programs are closed terms, we show that the compiler steps preserve the traceset of an open term e that is closed by substitution $[x \mapsto v]$. We formulate the necessary lemmas and sketch the proofs – the detailed proof is in Appendix C.

LEMMA 1 (ASSOC CORRECT). $\{ p; q; [x \mapsto v] \text{assoc}(m) \} \approx \{ p; q; [x \mapsto v] m \}$

LEMMA 2 (MNF CORRECT). $\{ p; q; [x \mapsto v] \text{mnf}(m) \} \approx \{ p; q; [x \mapsto v] m \}$

LEMMA 3 (MNF' CORRECT). $\{ \text{init}_C(\text{mnf}'(d; b; m)) \} \approx \{ \text{init}_C(d; b; m) \}$

LEMMA 4 (COMP CORRECT). $\{ [x \mapsto v] \text{init}_A(\text{comp}(d; b; \text{trmp}(m))) \} \approx \{ \text{init}_A(d; b; \text{trmp}([x \mapsto v] m)) \}$

LEMMA 5 (COMP' CORRECT). $\{ [x \mapsto v] \text{init}_A(\text{comp}'(d; b; \text{trmp}(m))) \} \approx \{ \text{init}_A(d; b; \text{trmp}([x \mapsto v] m)) \}$

Proof sketch. Lemma 1–5 hold by chain of transitive trace equality relations. We show that a term is trace-equal to the same term after compilation, by evaluating the compiled program (\rightarrow^*) and the original program (\leftarrow^*) until configurations converge. In the inductive case, we can remove the current compiler step in redex position under traceset equality (\approx) since traces before and after applying the compiler step are equal by induction hypothesis.

An interesting case is the proof of *comp* for $P = d; b; \text{awaitCl}(e_0, () \rightarrow e_1)$. The compiler transforms the remote communication awaitCl_s into the use of a guard and a trampoline. The compiled program steps to the use of awaitCl_t , the source program to awaitCl_s . In the attacker relation \rightarrow_b , arbitrary clients can send arbitrary values with awaitCl_t , leading to additional traces compared to the ones permitted in the source program where communication is modeled by awaitCl_s . We observe that awaitCl_s generates the trace elements $\text{msg}(c, v)$, $\text{wr}(0, \text{sender}, c)$ for all v and that awaitCl_t generates the trace elements $\text{msg}(c', v)$, $\text{wr}(0, \text{sender}, c')$ for all v, c' , which differ for $c' \neq c$.

Compilation adds an assert expression (Fig. 11) evaluated after receiving a value from a client. The assert gets stuck for configurations that produce trace elements with $c' \neq c$, removing the traces of such configurations from the trace set, leaving only the traces where $c' = c$. Hence, the trace set before and after compilation is equal under attack.

4 IMPLEMENTATION

Prisma is embedded into Scala (the host language) with its features implemented as a source-to-source macro expansion.⁶

⁶The implementation entails 21 Scala files, 3 412 lines of Scala source code (non-blank, non-comment) licensed under Apache 2.0 Open Source. The compiler phases are macros that recurse over the Scala AST: (a) the guarding phase, (b) the “simplifying” phase (including MNF translation, CPS translation of terms, defunctionalisation), and (c) the translation phase of (a subset) of Scala expressions and types to a custom intermediate representation based on Scala case classes. The intermediate representation is translated to Solidity code and passed to the Solidity compiler (solc).

The backend generating Solidity code is well separated. One could disable the compilation step to Solidity in the compilation pipeline, e.g., to run distributed code on multiple JVMs instead. In this case, the “contract code” would be executed by one computer (the “server”), and other computers would run the “client code”.

The Scala runtime of Prisma contains the implementation of the serialisable datatypes, portable between Scala and the EVM (fixed-size arrays, dynamic arrays, unsigned integers of length of powers of two up to 256 bit). Our runtime wraps `web3j` [50] (for invoking transactions and interacting with the blockchain in general), `headlong` [72] (for serialisation/deserialisation in the Ethereum-specific serialisation format), as well as code to parse Solidity and Ethereum error messages and to translate them to Scala error messages.

5 EVALUATION

We evaluate Prisma along two research questions:

RQ1 *Does Prisma support the most common dApps scenarios?*

RQ2 *Do Prisma’s abstractions affect performance?*

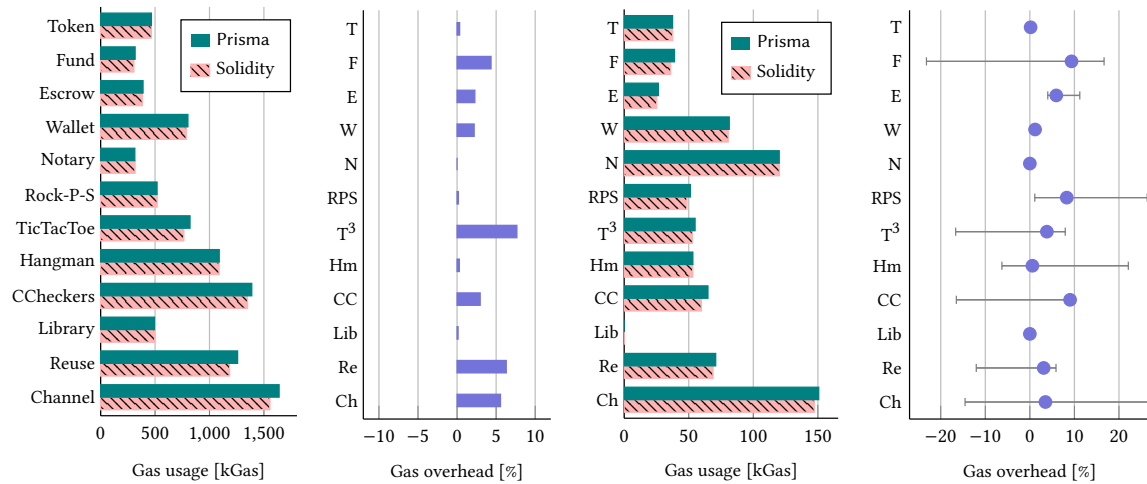
Case Studies and Expressiveness (RQ1). Five classes of smart contract applications have been identified [7]: Financial, Wallet, Notary, Game, and Library. To answer RQ1, we implemented at least one case study per category in Prisma. We implemented an ERC-20 Token,⁷ a Crowdfunding, and an Escrowing dApp as representatives of financial dApps. We cover *wallets* by implementing a multi-signature wallet, a special type of wallet that provides a transaction voting mechanism by only executing transactions, which are signed by a fixed fraction of the set of owners. We implemented a general-purpose *notary* contract enabling users to store arbitrary data, e.g., document hashes or images, together with a submission timestamp and the data owner. As *games*, we implemented TicTacToe (Section 2), Rock-Paper-Scissors, Hangman and Chinese Checkers. Rock-Paper-Scissors makes use of timed commitments [3], i.e., all parties commit to a random seed share and open it after all commitments have been posted. The same technique can be used to generate randomness for dApps in a secure way. To reduce expensive code deployment, developers outsource commonly used logic to library contracts. We demonstrate *library*-based development in Prisma by including a TicTacToe library to our case studies and another TicTacToe dApp which uses that library instead of deploying the logic itself.

We also implemented a state channel [29, 30, 57] for TicTacToe in Prisma, which is an example for the class of *scalability solutions* that have emerged more recently. State channels enable parties to move parts of their dApp to a non-blockchain consensus system, falling-back to the blockchain in case of disputes, thereby making the dApps more efficient where possible.

Our case studies are between 1 K and 7.5 K bytes which is a representative size: Smart contracts are not built for large-scale applications since the gas model limits the maximal computation and storage volumes and causes huge fees for complex applications. The median (average, lower quantile, upper quantile) of the bytecode size of distinct contracts deployed at the time of writing is at 4 K (5.5 K, 1.5 K, 7.5 K) [44]. We further elaborate on the case studies including a comparison of the lines of code in Prisma compared to the equivalent lines in Solidity and Javascript in Appendix A. Our case studies demonstrate that Prisma supports most common dApps scenarios.

Performance of Prisma DApps (RQ2). Performance on the Ethereum blockchain is usually measured in terms of an Ethereum-specific metric called *gas*. Each instruction of the Ethereum Virtual Machine (EVM) consumes gas which needs to be paid for by the users in form of transaction fees credited to

⁷A study investigating all blocks mined until Sep 15, 2018 [62], found that 72.9% of the high-activity contracts are token contracts compliant to ERC-20 or ERC-721, with an accumulated market capitalization of 12.7 B USD.



(a) Gas usage per deployment. (b) Gas overhead per deployment. (c) Gas usage per interaction. (d) Gas overhead per interaction.

Fig. 17. The cost of abstraction. Gas overhead of contracts written with Prisma vs. Solidity. (The right plot displays minima, averages, maxima.)

the miner. We refer to the Ethereum yellow paper [91] for an overview of the gas consumption of the different EVM instructions. To answer RQ2, we implement our case studies in both Prisma and in Solidity/JavaScript and compare their gas consumption. Unlike prior work, we do not model a custom gas structure, but consider the real EVM gas costs [90].

Experimental setup. We execute each case study on different inputs to achieve different execution patterns that cover all contract functions. Each contract invocation that includes parameters with various sizes (e.g., dynamic length arrays) is executed with a range of realistic inputs, e.g., for Hangman, we consider several words (2 to 40 characters) and different order of guesses, covering games in which the guesser wins and those in which they lose. Prisma and Solidity/JavaScript implementations are executed on the same inputs.

We perform the measurements on a local setup. As the execution in the Ethereum VM is deterministic, a single measurement suffices. We set up the local Ethereum blockchain with *Ganache* (Core v2.13.2) on the latest supported hard fork (Muir Glacier). All contracts are compiled to EVM byte code with *solc* (v0.8.1, optimized on 20 runs). We differentiate contract deployment and contract interaction. Deployment means uploading the contract to the blockchain and initializing its state, which occurs just once per application instance. A single instance typically involves several contract interactions, i.e., transactions calling public contract functions.

Results. Fig. 17 shows the average gas consumption of contract deployment (Fig. 17a) and interaction (Fig. 17c) as well as the relative overhead of Prisma vs. Solidity/JS of deployment (Fig. 17b) and interaction (Fig. 17d). As the gas consumption of contract invocations depends heavily on the executed function, the contract state, and the inputs, we provide the maximal, minimal and averaged overhead. The results show that the average gas consumption of Prisma is close to the one of Solidity/JS. Our compiler achieves a deployment overhead of maximally 6% (TicTacToe) or 86 K gas (TicTacToe Channel). The interaction overhead is below 10% for all case studies which at most amounts to 3.55 K gas.⁸

Prisma’s deployment overhead is mainly due to the automated flow control. To guarantee correct execution, Prisma manages a state variable for dApps with more than one state. The storage reserved for and the code deployed to maintain the state variable cause a constant cost of around

⁸equals 0.59 USD based on gas price and exchange course of April 15, 2021

Table 18. Related work.

Language	Encoding	Perspective	Protocol
Solidity	FSM	Local	Assertions
Obsidian	FSM	Local	Type states
Nomos	MNF	Local	Session types
Prisma	DS	Global	Control flow

45 K gas. In Solidity, developers manually check whether flow control is needed and, if so, may derive the state from existing contract variables to avoid a state variable if possible.

The Token, Notary, Wallet and Library case studies do not require flow control: each function can be called by any client at any time. Hence, their overhead is small. Escrow, Hangman and Rock-Paper-Scissors require a state variable, also in Solidity – which partially compensates the overhead of Prisma’s automated flow control. Crowdfunding, Chinese Checkers, TicTacToe (Library and Channel) do not require an explicit state variable in Solidity, as the state can be derived from the contract variables, e.g., the number of moves. Thus, these case studies have the largest deployment overhead.

While the average relative interaction overhead is constantly below 10 %, some contract invocations are far above, e.g., in Crowdfunding, TicTacToe Channel, and Rock-Paper-Scissors. Yet, case studies with such spikes also involve interactions that are executed within the same dApp instance with a negative overhead and amortize the costs of more costly transactions. These deviations are also mainly due to automated flow control. In EVM, setting a zero variable to some non-zero value costs more gas (20 K gas) than changing its value (5 K gas) [90], and setting the value to zero saves gas. Occupying and releasing storage via the state variable can cost or save gas in a different way than in traditional dApps without an explicit state variable, leading to different (and even negative) overhead in different transactions.

Besides the gas-overhead, we also consider the time-overhead of Prisma. In Ethereum, the estimated confirmation time for transactions is 3-5 minutes (assuming no congestion), which makes the number of on-chain interactions dominate the total execution time. As Prisma preserves the number of on-chain interactions, we assess the time-overhead of Prisma, if any, to be negligible.

Note that per se it is not possible to achieve a better gas consumption in Prisma than in Solidity – every contract compiled from Prisma can be implemented in Solidity. Given the abstractions we offer beyond the traditional development approach, and the sensibility of smart contracts to small changes in instructions, we conclude that our abstractions come with acceptable overhead. We are confident that further engineering effort can eliminate the observed overhead.

Threats to validity. The main threat is that the manually written code may be optimized better or worse than the code generated by the compiler. We mitigate this threat by applying all gas optimizations, our compiler performs automatically, to the Solidity implementations. An external threat is that changes in the gas pricing of Ethereum may affect our evaluation. For reproducibility, we state the Ethereum version (hard fork), we used in the paper.

6 DISCUSSION AND RELATED WORK

6.1 Smart Contract Languages for Enforcing Protocols

We compare Prisma to Solidity, Obsidian [18–20], and Nomos [25, 26]. We highlight these languages as those also address the correctness of the client–contract interactions. Tab. 18 overviews the features of the surveyed languages for (a) the *perspective* of defining interacting parties, (b) the used *encoding* of the interaction effects, and (c) the method used to check the contract-client interaction

```

1  asset contract TTT {
2    state Funding{}; state Executing{}; state Finished{}; state Closed{}
3    transaction Fund(TTT@Funding>>(Funding|Executing) this, int c) {
4      /*...*/; if (/* enough funds? */) -> Executing else -> Funding }
5    transaction Move(TTT@Executing>>(Executing|Finished) this, int x, int y) {
6      /*...*/; if (/* game over? */) -> Finished else -> Executing }
7    transaction Payout(TTT@Finished>>Closed this) {
8      /*...*/; -> Closed } }

```

Fig. 19. Obsidian.

```

1  type Funding = int -> +{ notenough: Funding, enough: Executing }
2  type Executing = int -> int -> +{ notdone: Executing, done: Finished }
3  type Finished = int -> 1
4  proc contract funding : . |{*}- ($s : Funding) = {
5    a = recv $s ; /* ... */
6    if /* enough funds? */ then $s .notenough; $s <- funding
7      else $s .enough; $s <- executing }
8  proc contract executing : . |{*}- ($s : Executing) = {
9    x = recv $s ; y = recv $s ; /* ... */
10   if /* game over? */ then $s .notdone; $s <- executing
11   else $s .done; z = recv $s ; close $s }

```

Fig. 20. Nomos.

$$\begin{array}{c}
\text{NOMOSR} \\
\frac{\Psi; \Gamma, (y:A) \vdash P :: (c : B)}{\Psi; \Gamma \vdash (y \leftarrow \text{recv } c; P) :: (c : A \multimap B)} \\
\\
\text{NOMOS} \\
\frac{\Psi; \Gamma \vdash P :: (c : B)}{\Psi; \Gamma, (w:A) \vdash (\text{send } c \ w; P) :: (c : A \otimes B)} \\
\\
\text{OBSIDIAN} \\
\frac{(\text{transaction } T \ m(\overline{t.(s \gg s')} \ x)\{\dots\}) \in \text{members}_{t_0}}{\Delta, \overline{e:t.s} \vdash e_0.m(\overline{e}) : T \dashv \Delta, \overline{e:t.s'}}
\end{array}$$

Fig. 21. Excerpts of simplified Nomos and Obsidian typing rules.

protocol. Fig. 4c, 19, and 20 show code snippets in these languages, each encoding the *TicTacToe* state machine from Fig. 1. All three languages focus solely on the contract and do not state how clients are developed, hence only contract code is shown.

All three approaches take a **local perspective** on interacting parties: Contract and clients are defined separately, and their interaction is encoded by explicit send and receive side effects. In Solidity and Obsidian, receive corresponds to arguments and send to return values of methods defined in the contract classes. In Nomos, send and receive are expressed as procedures operating over a channel – given a channel c , sending and receiving is represented by explicit statements ($x = \text{recv } c; \dots$ and $\text{send } c \ x; \dots$).

The approaches differ in the **encoding style** of communication effects. Solidity and Obsidian adopt an *FSM-style encoding*: Contract fields encode states, methods encode transitions. The contract in Fig. 4c represents FSM states via the phase field with initial state `Funding` (Line 30). The `Fund`, `Move` and `Payout` methods are transitions, e.g., `Payout` transitions the contract into the final state `Closed` (Line 51). The FSM-style encoding results in an implicitly-everywhere concurrent programming

model, which is complex to reason about and unfitting for dApps because the execution model of blockchains is inherently sequential – all method invocations are brought into a world-wide total order. Nomos adopts the *monadic normal form (MNF)* via do-notation to order effects. While the implementation of TicTacToe in FSM style requires three methods (Fund, Move, Payout – one per transition), we only need two methods in MNF-style (funding, executing – one per state with multiple entry points), and a single method in DS-style (init). For instance, the sequence of states and transitions $Executing \xrightarrow{Move(x,y)} Finished \xrightarrow{Payout()} Closed$ in Nomos can be written sequentially in do-notation by inlining the last function which only has a single entry point. Still, do-notation can be cumbersome (e.g., funding and executing in Nomos are separate methods that cannot be inlined since they have multiple entry points and model loops).

All three languages require an **explicit protocol** for governing the send–receive interactions, to ensure that every send effect has a corresponding receive effect in an interacting – separately defined – party. In Solidity, developers express the protocol via run-time assertions to guard against invoking the methods in an incorrect order (e.g., `require(phase==Finished)` in Fig. 4c, Line 40). Unlike Solidity, which does not support statically checking protocol compliance, Nomos and Obsidian employ *behavioral typing* for static checks. Deployed contracts may interact with third-party, potentially manipulated clients. Compile-time checking alone cannot provide security guarantees. Yet, complementing run-time enforcement with static checks helps detecting cases that are guaranteed to fail at run time ahead of time.

Obsidian. Obsidian employs typestates to increase safety of contract–client communication. Contracts define a number of typestates; A method call can change the typestate of an object, and calling a method on a receiver that is in the wrong typestate results in a typing error. Each method in Fig. 19 is annotated with the state in which it can be called, e.g., Payout requires state Finished, and transitions to closed (Line 7).

Nomos. Nomos employs session types. The session types Funding, Executing, Finished in Fig. 20 encode the protocol. Receiving a message is represented by a function type, e.g., in the Funding state, we receive an integer `int -> ...` (Line 1). We respond by either repeating the funding (Funding), or continuing to the next state of the protocol (Executing). This is represented by internal choice `+{ ... }` that takes multiple possible responses giving each of them a unique label (notenough and enough). Type 1 indicates the end of a protocol (Line 3). The contract processes funding (Line 4) and executing (Line 8) implement the protocol. The `recv` operation (Line 5) takes a session-typed channel of form $\tau \rightarrow u$, returns a value of type τ and changes the type of the channel to u . A session type for internal choice `+{ ... }`, requires the program to select one of the offered labels (e.g., `$s.notenough` in Line 6 and `$s.enough` in Line 7), e.g., in the left and right branch of a conditional statement.

Type systems. We show excerpts of simplified typing rules for Nomos and Obsidian (Fig. 21). Nomos rules have the form $\Psi; \Gamma \vdash P :: (c:A)$. A process P offers a channel c of type A with values in context Ψ and channels in Γ . We can see that variables change their type to model the linearity of session types in the NOMOS (and NOMOSR) rule: Sending (and receiving) changes the type of the channel c from $A \multimap B$ to B (and $A \otimes B$ to B). Obsidian rules have the form $\Delta \vdash e:t \dashv \Delta'$. An expression e has type t in context Δ and changes Δ to Δ' . We can see that variables change their type on method invocation (OBSIDIAN): A method m in class t_0 with arguments e_i of type t_i , returning T , changes the type state of the arguments from s_i to s'_i . For Prisma, instead, a standard judgement $\Gamma \vdash e : T$ suffices for communication. Variables do not change their type. `awaitCl(p){b}` has type T in context Γ if p is a predicate of *Addr* and b is a pair of *Ether* and T :

$$\frac{\text{PRISMA} \quad \Gamma \vdash p : \text{Addr} \rightarrow \text{Bool} \quad \Gamma \vdash b : \text{Ether} \times T}{\Gamma \vdash \text{awaitCl}(p)\{b\} : T}$$

Prisma. As shown in Tab. 18, Prisma occupies an unexplored point in the design space: *global* instead of local perspective on interacting parties, *direct style (DS)* instead of FSM or MNF encoding of effects, and *control flow* instead of extra protocol for governing interactions.

Prisma takes a **global perspective** on interacting parties. The parties execute the same program, where pairs of send and receive actions that “belong together” are encapsulated into a single **direct-style** operation, which is executed differently by sending and receiving parties. Hence, dApps are modeled as sequences and loops of send-receive-instructions shared by interacting parties. Due to the global direct style perspective, it is syntactically impossible to define parties with mismatching send and receive pairs. Hence, a standard System-F-like type system suffices. The interaction protocol follows directly from the sequential **control flow** of interaction points in the program – the compiler can automatically generate access and control guards with correctness guarantees. Semantically, Prisma features a by-default-sequential programming model, intentionally making the sequential execution of methods explicit, including interaction effects.

The global direct-style model also leads to improved design of dApps: No programmatic state management on the contract and no so-called *callback hell* [31] on the client. The direct style is also superior to Nomos’ MNF style. The tierless model avoids boilerplate: Client code can directly access public contract variables, unlike JavaScript code, which has to access them via a function call that requires either an await expression or a callback.⁹ Additionally, the developer has to implement getters for public variables with complex data types such as arrays.¹⁰ We provide some code measurements (lines of code and number of cross-tier control-flow calls) of our Prisma and Solidity/JS dApp case studies in Appendix B.

Finally, using one language for both the contract and the clients naturally enables static type safety of values that cross the contract–client boundary: an honest, non-compromized client cannot provide inconsistent input, e.g., with wrong number of parameters or falsely encoded types.¹¹ In a setting with different language stacks, it is not possible to statically detect type mismatches in the client–contract interaction; e.g., Solidity has a type *bytes* for byte arrays, which does not exist in JavaScript (commonly used to implement clients of a Solidity contract). Client developers need to encode byte arrays using hexadecimal string representations starting with “0x”, otherwise they cannot be interpreted by the contract.

6.2 Other Related Work

Smart contract languages. Harz and Knottenbelt [45] survey smart contract languages, Hu et al. [48] survey smart contract tools and systems, Wöhrer and Zdun [89] give an overview of design patterns in smart contracts. Brünjes and Gabbay [13] distinguish between imperative and functional smart contract programming. *Imperative contracts* are based on the account model; the most prominent language is Solidity [32]. *Functional* ones [14, 77, 78] are based on EUTxO (Extended Unspent Transaction Output) model [39]. State channels [15, 29, 30, 57] optimistically optimize contracts for the functional model. Prisma does not yet support compilation to state channels but we plan to treat them as another kind of tier.

⁹Obsidian and Nomos do not provide any client design, so we can only compare to Solidity/JavaScript.

¹⁰For simple data types the getter is generated automatically.

¹¹Recall that in dApps checking cross-tier type-safety is not a security feature but a design-time safety feature (due to the open-world assumption of the execution model of public ledgers).

Smart contracts as state machines. Scilla [79] is an automata-based compiler for contracts. FSolidM [55] enables creating contracts via a graphical interface. VeriSolid [56] generates contracts from graphical models enriched with predicates based on computational tree logic. EFSM tools [84] generate contracts from state machines and linear temporal logic. Prisma avoids a separate specification but infers transactions and their order from the control flow of a multitier dApp.

Analysis tools. Durieux et al. [28] and Ferreira et al. [35] empirically validate languages and tools and relate design patterns to security vulnerabilities, extending the survey by Di Angelo and Salzer [4]. Our work is complementary, targeting the correctness of the distributed program flow. For vulnerabilities not related to program flow (e.g., front-running, or bad randomness), developers (using Solidity/JavaScript or Prisma) can use the surveyed analysis tools.

Multitier languages. Multitier programming was pioneered by Hop [80, 81]. Modeling a persistent session in client–server applications with continuations was mentioned by Queinnec [69] and elaborated in Links [22, 38]. Eliom [70] supports bidirectional client–server communication for web applications. ScalaLoci [87] generalizes the multitier model to generic distributed architectures. Our work specializes it to the dApp domain and its specific properties. Giallorenzo et al. [41] establish interesting connections between multitier (subjective) and choreographic (objective) languages – two variants of the global model. Prisma adopts the subjective view, which naturally fits the dApp domain, where a dominant role (contract) controls the execution and diverts control to other parties (clients) to collect their input.

Mashic [51] is a compiler for a *mashup* between two JavaScript programs: the untrusted embedded (iframe) *gadget(s)* and the trustworthy hosting *integrator* program, which communicate via messages. The authors prove that the compiler guarantees integrity and confidentiality. More specifically, the gadget(s) cannot learn more than what the integrator sends and, analogously, the gadget’s influence is limited to the integrator’s interface. In Mashic, the two programs are separate and the compiler checks that they communicate only via specified messages. In contrast, in Prisma, client and contract code are mixed. Thus, in addition to checking that only the specified messages are used, we can also check the interaction protocol – expressed by the structure of the control flow of the program – and ensure that it is followed by the target program after compilation.

Swift’s [17] secure automatic partitioning approach uses information flow policies to derive placements. Based on the policies, a constraint solver with integer programming heuristically picks a placement such that network traffic is minimal and information flow integrity is preserved. In contrast, placements in Prisma are explicit to the developer. Further, in blockchain programming, every single instruction generated by the compiler potentially incurs high costs. Therefore, we demonstrated that our compiler generates inexpensive programs, whereas Swift does not consider the program’s execution cost.

Effectful programs and meta-programming. MNF and CSP are widely discussed as intermediate compiler forms [6, 21, 37, 49, 54]. F# computation expressions [64] support control-flow operators in monadic expressions. OCaml supports a monadic and applicative `let` [88]: more flexible than `do`-notation but still restricted to MNF. Idris’ `!`-notation [10] inspired the GHC proposal for monadic inline binding [68]. Scala supports effectful programs through coroutines [67], `async/await` [73], monadic inline binding [11], `Dsl.scala` [92] and a (deprecated) compiler plugin for CPS translation [74]. The `dotty-cps-async` macro [82] supports `async/await` and similar effects for the Dotty compiler.

7 CONCLUSION

We proposed Prisma, the first global language for dApps that features direct style communication. Compared to the state of the art, Prisma (a) enables the implementation of contract and client logic within the same development unit, rendering intricacies of the heterogeneous technology stack obsolete and avoiding boilerplate code, (b) provides support for explicitly encoding the intended program flow and (c) reduces the risk of human failures by enforcing the intended program flow and forcing developers to specify access control.

Unlike previous work that targeted challenges in the development of dApps with advanced typing disciplines e.g., session types, our model does not exhibit visible side effects and gets away with a standard System-F-style type system. We describe the design and the main features of Prisma informally, define its formal semantics, formalize the compilation process and prove it correct. We demonstrate Prisma’s applicability via case studies and performance benchmarks.

We plan to generate state channels – to optimistically cost-optimize dApps – similar to how we generate state machines from high-level logic. Further, we believe that our technique for deriving the communication protocol from direct-style control flow generalizes beyond the domain of smart contracts and we will explore its further applicability.

ACKNOWLEDGMENTS

This work has been funded by the German Federal Ministry of Education and Research iBlockchain project (BMBF No. 16KIS0902), by the German Research Foundation (DFG, SFB 1119 – CROSSING Project), by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, by the Hessian LOEWE initiative (emergenCITY), by the Swiss National Science Foundation (SNSF, No. 200429), and by the University of St. Gallen (IPF, No. 1031569).

Thanks to George Zakhour for feedback on the initial draft.

REFERENCES

- [1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1 (2009), 4:1–4:40. <https://doi.org/10.1145/1609956.1609960>
- [2] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA.
- [3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure Multiparty Computations on Bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 443–458. <https://doi.org/10.1109/SP.2014.35>
- [4] Monika Di Angelo and Gernot Salzer. 2019. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *IEEE International Conference on Decentralized Applications and Infrastructures, DAPPCON 2019, Newark, CA, USA, April 4-9, 2019*. IEEE, 69–78. <https://doi.org/10.1109/DAPPCON.2019.00018>
- [5] Monika Di Angelo and Gernot Salzer. 2020. Wallet Contracts on Ethereum. *CoRR* abs/2001.06909 (2020). arXiv:2001.06909 <https://arxiv.org/abs/2001.06909>
- [6] Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press.
- [7] Massimo Bartoletti and Livio Pompianu. 2017. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International conference on financial cryptography and data security*. Springer, 494–509.
- [8] Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause ‘n’ Play: Formalizing Asynchronous C#. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 233–257. https://doi.org/10.1007/978-3-642-31057-7_12
- [9] Sam Blackshear, Evan Cheng, D. Dill, Victor Gao, B. Maurer, T. Nowacki, Alistair Pott, S. Qadeer, Dario Russi, Stephane Sezer, Tim Zakian, and Run tian Zhou. 2019. Move: A Language With Programmable Resources. <https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2019-06-18.pdf>.
- [10] Edwin Brady. 2007. The Idris Tutorial. Interfaces. Monads and do-notation. !-notation. <http://docs.idris-lang.org/en/latest/tutorial/interfaces.html#notation>. Accessed 14-11-2020.

- [11] Flavio W. Brasil and Sameer Brenn. 2017. Monadless – Syntactic sugar for monad composition in Scala. <https://github.com/monadless/monadless>.
- [12] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. 2016. An In-Depth Look at the Parity Multisig Bug. hackingdistributed.com/2017/07/22/deep-dive-parity-bug/. Accessed 14-11-2020.
- [13] Lars Brünjes and Murdoch James Gabbay. 2020. UTxO- vs Account-Based Smart Contract Blockchain Programming Paradigms. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 12478)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 73–88. https://doi.org/10.1007/978-3-030-61467-6_6
- [14] Manuel Chakravarty, Roman Kireev, Kenneth MacKenzie, Vanessa McHale, Jann Müller, Alexander Nemish, Chad Nester, Michael Peyton Jones, Simon Thompson, Rebecca Valentine, and Philip Wadler. 2019. Functional Blockchain Contracts. (2019). <https://iohk.io/en/research/library/papers/functional-blockchain-contracts/>
- [15] Manuel M. T. Chakravarty, Sandro Coretti, Matthias Fitzi, Peter Gazi, Philipp Kant, Aggelos Kiayias, and Alexander Russell. 2020. Hydra: Fast Isomorphic State Channels. *IACR Cryptol. ePrint Arch.* 2020 (2020), 299. <https://eprint.iacr.org/2020/299>
- [16] Kwanghoon Choi and Byeong-Mo Chang. 2019. A Theory of RPC Calculi for Client–Server Model. *Journal of Functional Programming* 29 (2019). <https://doi.org/10.1017/S0956796819000029>
- [17] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure web applications via automatic partitioning. In *Symposium on Operating Systems Principles*.
- [18] Michael J. Coblenz. 2017. Obsidian: a safer blockchain programming language. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 97–99. <https://doi.org/10.1109/ICSE-C.2017.150>
- [19] Michael J. Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Jonathan Aldrich, Joshua Sunshine, and Brad A. Myers. 2019. User-Centered Programming Language Design in the Obsidian Smart Contract Language. *CoRR* abs/1912.04719 (2019). arXiv:1912.04719 <http://arxiv.org/abs/1912.04719>
- [20] Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. *CoRR* abs/1909.03523 (2019). arXiv:1909.03523 <http://arxiv.org/abs/1909.03523>
- [21] Youyou Cong, Leo Oswald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.* 3, ICFP (2019), 79:1–79:28. <https://doi.org/10.1145/3341643>
- [22] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects (Amsterdam, The Netherlands) (FMCO’06)*. Springer-Verlag, Berlin, Heidelberg, 266–296. <http://dl.acm.org/citation.cfm?id=1777707.1777724>
- [23] Phil Daian. 2016. Analysis of the DAO exploit. <https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>. Accessed 14-11-2020.
- [24] Dapp.com. 2020. 2020 Q2 Dapp Market Report. <https://www.dapp.com/article/q2-2020-dapp-market-report>.
- [25] Ankush Das, S. Balzer, J. Hoffmann, and F. Pfenning. 2019. Resource-Aware Session Types for Digital Contracts. *ArXiv* abs/1902.06056 (2019).
- [26] Ankush Das, Jan Hoffmann, and Frank Pfenning. 2021. Nomos: A Protocol-Enforcing, Asset-Tracking, and Gas-Aware Language for Smart Contracts. (2021).
- [27] Mariangiola Dezani-Ciancaglini and Ugo de’Liguoro. 2009. Sessions and Session Types: An Overview. In *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6194)*, Cosimo Laneve and Jianwen Su (Eds.). Springer, 1–28. https://doi.org/10.1007/978-3-642-14458-5_1
- [28] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts. In *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 530–541. <https://doi.org/10.1145/3377811.3380364>
- [29] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. 2019. Multi-party Virtual State Channels. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11476)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, 625–656. https://doi.org/10.1007/978-3-030-17653-2_21
- [30] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. 2018. General State Channel Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 949–966.

- <https://doi.org/10.1145/3243734.3243856>
- [31] Jonathan Edwards. 2009. Coherent reaction. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA*, Shail Arora and Gary T. Leavens (Eds.). ACM, 925–932. <https://doi.org/10.1145/1639950.1640058>
 - [32] Ethereum Foundation. 2015. Solidity Documentation. <https://docs.soliditylang.org/en/v0.8.1/>. Accessed 14-11-2020.
 - [33] Ethereum Foundation. 2015. Solidity Documentation – Common Patterns. <https://docs.soliditylang.org/en/v0.7.4/common-patterns.html>. Accessed 14-11-2020.
 - [34] Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (1992), 235–271. [https://doi.org/10.1016/0304-3975\(92\)90014-7](https://doi.org/10.1016/0304-3975(92)90014-7)
 - [35] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. *CoRR* abs/2007.04771 (2020). arXiv:2007.04771 <https://arxiv.org/abs/2007.04771>
 - [36] Klint Finley. 2016. A \$50 Million Hack Just Showed That the DAO Was All Too Human. *Wired* (6 2016).
 - [37] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23–25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. <https://doi.org/10.1145/155090.155113>
 - [38] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 28 (Jan. 2019), 29 pages. <https://doi.org/10.1145/3290341>
 - [39] Murdoch James Gabbay. 2020. What is an EUTxO blockchain? *CoRR* abs/2007.12404 (2020). arXiv:2007.12404 <https://arxiv.org/abs/2007.12404>
 - [40] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2020. Choreographies as Objects. arXiv:2005.09520 [cs.PL]
 - [41] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty Languages: The Choreographic and Multitier Cases (Pearl). In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11–17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:27. <https://doi.org/10.4230/LIPIcs.ECOOP.2021.22>
 - [42] PolyCrypt GmbH. 2020. Perun Network. <https://perun.network>.
 - [43] Google Inc. 2021. Google Cloud BigQuery: Contract deployment per month. <https://console.cloud.google.com/bigquery>. Query: SELECT EXTRACT(MONTH FROM c.block_timestamp) AS m, EXTRACT(YEAR FROM c.block_timestamp) AS y, COUNT(c.address) FROM ‘bigquery-public-data.ethereum_blockchain.live_contracts’ AS c GROUP BY m, y ORDER BY y, m; Accessed 07-07-2021.
 - [44] Google Inc. 2021. Google Cloud BigQuery: Contract size. <https://console.cloud.google.com/bigquery>. Query: WITH d as (SELECT DISTINCT c.bytecode,(LENGTH(c.bytecode)-2)/2 as s FROM ‘[...]live_contracts’ AS c) SELECT PERCENTILE_CONT(d.s, [0, 0.25, 0.5, 0.75, 1]) OVER () AS M FROM d LIMIT 1; Accessed 14-11-2021.
 - [45] Dominik Harz and William J. Knottenbelt. 2018. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. *CoRR* abs/1809.09805 (2018). arXiv:1809.09805 <http://arxiv.org/abs/1809.09805>
 - [46] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. <https://doi.org/10.1145/359576.359585>
 - [47] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneswar, India, February 9–12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6536)*, Raja Natarajan and Adegboyega K. Ojo (Eds.). Springer, 55–75. https://doi.org/10.1007/978-3-642-19056-8_4
 - [48] Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. 2020. A Comprehensive Survey on Smart Contract Construction and Execution: Paradigms, Tools and Systems. *CoRR* abs/2008.13413 (2020). arXiv:2008.13413 <https://arxiv.org/abs/2008.13413>
 - [49] Andrew Kennedy. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1–3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 177–190. <https://doi.org/10.1145/1291151.1291179>
 - [50] Web3 Labs. 2016. Web3j: Web3 Java Ethereum Dapp API (GitHub Repository). <https://github.com/web3j/web3j>.
 - [51] Zhengqin Luo and Tamara Rezk. 2012. Mashic Compiler: Mashup Sandboxing Based on Inter-frame Communication. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25–27, 2012*, Stephen Chong (Ed.). IEEE Computer Society, 157–170. <https://doi.org/10.1109/CSF.2012.22>
 - [52] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (Vienna, Austria) (CCS ’16)*. Association for Computing Machinery, New York, NY, USA, 254–269. <https://doi.org/10.1145/2976749.2978309>

- [53] Simon Marlow. 2010. Haskell 2010: Language Report. Expressions. Do Expressions. <https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-470003.14>. Accessed 14-11-2020.
- [54] Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling without continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 482–494. <https://doi.org/10.1145/3062341.3062380>
- [55] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10957)*, Sarah Meiklejohn and Kazue Sako (Eds.). Springer, 523–540. https://doi.org/10.1007/978-3-662-58387-6_28
- [56] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11598)*, Ian Goldberg and Tyler Moore (Eds.). Springer, 446–465. https://doi.org/10.1007/978-3-030-32101-7_27
- [57] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. 2019. Sprites and State Channels: Payment Networks that Go Faster Than Lightning. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11598)*, Ian Goldberg and Tyler Moore (Eds.). Springer, 508–526. https://doi.org/10.1007/978-3-030-32101-7_30
- [58] Mix. 2019. These are the top 10 programming languages in blockchain. <https://thenextweb.com/hardfork/2019/05/24/javascript-programming-java-cryptocurrency/>. Accessed 14-11-2020.
- [59] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. 2014. Service-Oriented Programming with Jolie. In *Web Services Foundations*, Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel (Eds.). Springer, 81–107. https://doi.org/10.1007/978-1-4614-7518-7_4
- [60] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (San Juan, PR, USA) (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 653–663. <https://doi.org/10.1145/3274694.3274743>
- [61] Reed Oei, Michael J. Coblenz, and Jonathan Aldrich. 2020. Psamathe: A DSL with Flows for Safe Blockchain Assets. *CoRR* abs/2010.04800 (2020). arXiv:2010.04800 <https://arxiv.org/abs/2010.04800>
- [62] Gustavo A Oliva, Ahmed E Hassan, and Zhen Ming Jack Jiang. 2020. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empirical Software Engineering* 25 (2020), 1864–1904. Issue 3. <https://doi.org/10.1007/s10664-019-09796-5>
- [63] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6 (2019), 125:1–125:36. <https://doi.org/10.1145/3280984>
- [64] Tomas Petricek and Don Syme. 2014. The F# Computation Expression Zoo. In *PADL (Lecture Notes in Computer Science, Vol. 8324)*. Springer, 33–48.
- [65] Sergey Petrov. 2017. Another Parity Wallet hack explained. <https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c>
- [66] Andrew M. Pitts. 2000. Operational Semantics and Program Equivalence. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures (Lecture Notes in Computer Science, Vol. 2395)*, Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva (Eds.). Springer, 378–412. https://doi.org/10.1007/3-540-45699-6_8
- [67] Aleksandar Prokopec. 2015. Scala Coroutines. <https://github.com/storm-enroute/coroutines>.
- [68] Jon Purdy. 2017. Discussion on GHC Pre-Proposal: Add InlineBindings proposal. <https://github.com/ghc-proposals/ghc-proposals/pull/64>. Accessed 14-11-2020.
- [69] Christian Queinnec. 2000. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 23–33. <https://doi.org/10.1145/351240.351243>
- [70] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A core ML language for Tierless Web Programming. In *Proceedings of the 14th Asian Symposium on Programming Languages and Systems (Hanoi, Vietnam) (APLAS '16)*, Atsushi Igarashi (Ed.). Springer-Verlag, Berlin, Heidelberg, 377–397. https://doi.org/10.1007/978-3-319-47958-3_20
- [71] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *ACM '72*.
- [72] Evan Saulpaugh. 2018. Headlong (GitHub Repository). <https://github.com/esaulpaugh/headlong>.
- [73] Scala Development Team. 2012. scala-async. A Scala DSL to enable a direct style of coding when composing Futures. <https://github.com/scala/scala-async>.

- [74] Scala Development Team. 2013. scala-continuations. The Scala delimited continuations plugin and library. <https://github.com/scala/scala-continuations>.
- [75] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing safe smart contracts in Flint. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Nice, France, April 09-12, 2018*, Stefan Marr and Jennifer B. Sartor (Eds.). ACM, 218–219. <https://doi.org/10.1145/3191697.3213790>
- [76] Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. 2019. Flint for Safer Smart Contracts. *CoRR* abs/1904.06534 (2019). arXiv:1904.06534 <http://arxiv.org/abs/1904.06534>
- [77] Pablo Lamela Seijas, Alexander Nemish, David Smith, and Simon THompson. 2020. Marlowe: implementing and analysing financial contracts on blockchain, Tiziana Margaria and Bernhard Steffen (Eds.). *Workshop on Trusted Smart Contracts @ FC 2020*. <https://iohk.io/en/research/library/papers/marloweimplementing-and-analysing-financial-contracts-on-blockchain/>
- [78] Pablo Lamela Seijas and Simon J. Thompson. 2018. Marlowe: Financial Contracts on Blockchain. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISO LA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 11247)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 356–375. https://doi.org/10.1007/978-3-030-03427-6_27
- [79] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *PACMPL* 3, OOPSLA (2019), 185:1–185:30. <https://doi.org/10.1145/3360611>
- [80] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop, A Language for Programming the Web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, OR, USA) (*OOPSLA Companion '06*). ACM, New York, NY, USA.
- [81] Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) (*ICFP '16*). ACM, New York, NY, USA, 180–192. <https://doi.org/10.1145/2951913.2951916>
- [82] Ruslan Shevchenko. 2020. dotty-cps-async. <https://github.com/rssh/dotty-cps-async>.
- [83] State Channels contributors. 2020. State Channels. <https://statechannels.org/>.
- [84] Dmitrii Suvorov and Vladimir Ulyantsev. 2019. Smart Contract Design Meets State Machine Synthesis: Case Studies. *CoRR* abs/1906.02906 (2019). arXiv:1906.02906 <http://arxiv.org/abs/1906.02906>
- [85] Uniswap Labs. 2021. Uniswap Info. <https://v2.info.uniswap.org/home>. Accessed 07-07-2021.
- [86] Philip Wadler. 2012. Propositions as sessions. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 273–286. <https://doi.org/10.1145/2364527.2364568>
- [87] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. <https://doi.org/10.1145/3276499>
- [88] Leo White. 2018. OCaml: Add "monadic" let operators. <https://github.com/ocaml/ocaml/pull/1947>.
- [89] Maximilian Wöhrer and Uwe Zdun. 2020. From Domain-Specific Language to Code: Smart Contracts and the Application of Design Patterns. *IEEE Softw.* 37, 4 (2020), 37–42. <https://doi.org/10.1109/MS.2020.2993470>
- [90] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. <https://ethereum.github.io/yellowpaper/paper.pdf>. Ethereum Yellow Paper: A Formal Specification of Ethereum, a Programmable Blockchain. BERLIN VERSION 0e0eba8 – 2021-11-02. Accessed 14-11-2020.
- [91] Gavin Wood. 2022. Ethereum Yellow Paper. <https://ethereum.github.io/yellowpaper/paper.pdf>.
- [92] Bo Yang. 2016. Dsl.scala – A framework to create embedded Domain-Specific Languages in Scala. <https://github.com/ThoughtWorksInc/Dsl.scala>.

A CASE STUDIES

This section describes the implemented case studies in detail. Bartoletti and Pompianu [7] identify five classes of smart contract applications: Financial, Notary, Game, Wallet, and Library. Our case studies include at least one application per category (Table 22). In addition, we consider scalability solutions.

Financial. These apps include digital tokens, crowdfunding, escrowing, advertisement, insurances and sometimes Ponzi schemes. A study investigating all blocks mined until September 15th, 2018 [62], found that 72.9% of the high-activity contracts are token contracts compliant to ERC-20 or ERC-721, which have an accumulated market capitalization of US \$ 12.7 billion. We have implemented a fungible Prisma token of the ERC-20 standard. Further, we implemented crowdfunding and escrowing case studies. These case studies demonstrate how to send and receive coins with Prisma, which is the basic functionality of financial applications. Other financial use cases can be implemented in Prisma with similar techniques.

Notary. These contracts use the blockchain to store data immutably and persistently, e.g., to certify their ownership. We implemented a general-purpose notary contract enabling users to store arbitrary data, e.g., document hashes or images, together with a submission timestamp and the data owner. This case study demonstrates that Notaries are expressible with Prisma.

Games. We implemented TicTacToe (Section 2), Rock-Paper-Scissors, Hangman and Chinese Checkers. Hangman evolves through multiple phases and hence benefits from the explicit control flow definition in Prisma more than the other game case studies. The game Chinese Checkers is more complex than the others, in regard to the number of parties, the game logic and the number of rounds, and hence, represents larger applications. Rock-Paper-Scissors illustrates how randomness for dApps is securely generated. Every Ethereum transaction, including the executions of contracts, is deterministic – all participants can validate the generation of new blocks. Hence, secure randomness is negotiated among parties: in this case, by making use of timed commitments [3], i.e., all parties commit to a random seed share and open it after all commitments have been posted. The contract uses the sum of all seed shares as randomness. If one party aborts prior to opening its commitment, it is penalized. In Rock-Paper-Scissors both parties commit to their choice – their random share – and open it afterwards. Other games of chance, e.g., gambling contracts, use the same technique.

Wallet. A wallet contract manages digital assets, i.e., cryptocurrencies and tokens, and offers additional features such as shared ownership or daily transaction limits. At August 30, 2019, 3.9 M of 17.9 M (21%) deployed smart contracts have been different types of wallet contracts [5]. Multi-signature wallets are a special type of wallet that provides a transaction voting mechanism by only executing transactions, which are signed by a fixed fraction of the set of owners. Wallets transfer money and call other contracts in their users stead depending on run-time input, demonstrating calls among contracts in Prisma. Further, a multi-signature wallet uses built-in features of the Ethereum VM for signature validation, i.e., data encoding, hash calculation, and signature verification, showing that these features are supported in Prisma.

Libraries. As the cost of deploying a contract increases with the amount of code in Ethereum, developers try to avoid code repetitions. Contract inheritance does not help: child contracts simply copy the attributes and functions from the parent. Yet, one can outsource commonly used logic to *library contracts* that are deployed once and called by other contracts. For example, the TicTacToe dApp and the TicTacToe channel in our case studies share some logic, e.g., to check the win condition. To demonstrate libraries in Prisma, we include a TicTacToe library to our case studies and another on-chain executed TicTacToe dApp which uses such library instead of deploying the logic itself. Libraries use a call instruction similar to wallets, although the call target is typically known at deployment and can be hard-coded.

Fig. 22. Categories and Cross-tier calls.

Category	Case study	Cross-tier calls	Prisma LoC	Solidity + JavaScript LoC
Financial	Token	4	79	48 + 50
	Crowdfunding	11	59	27 + 63
	Escrow	9	63	33 + 56
Wallet	Multi-signature wallet	3	76	41 + 52
Notary	General-purpose notary	3	32	16 + 36
Game	Rock Paper Scissors	12	79	41 + 77
	TicTacToe	5	61	31 + 52
	Hangman	15	119	86 + 83
	Chinese Checkers	4	167	141 + 47
Library	TicTacToe library	–	167	141 + –
	TicTacToe using library	5	53	29 + 52
Scalability	TicTacToe channel	9	177	56 + 177

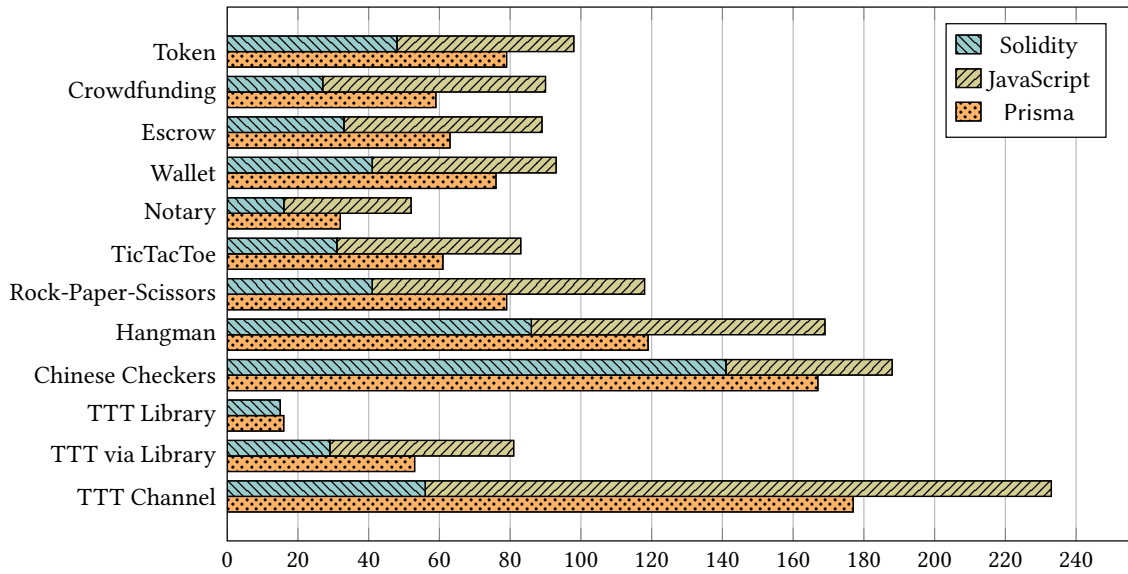


Fig. 23. LOC in Solidity/JavaScript and Prisma.

Scalability solutions. State channels [29, 30, 57] are scalability solutions, which enable a fixed group of parties to move their dApp to a non-blockchain consensus protocol: the execution falls-back to the blockchain in case of disputes. Similar to multi-signature wallets, state channels use built-in signature validation. We implemented a state channel for TicTacToe¹² to demonstrate that Prisma supports state channels.

B EMPIRICAL EVALUATION OF DESIGN QUALITY

In Section 6, we argued that with Prisma, (a) we provide communication safety with a standard system-F-like type-system, (b) the program flow can be defined explicitly and is enforced automatically, (c) dApp developers need to master a single technology that covers both tiers, (d) cross-tier type-safety can be checked at compile-time, and (e) the code is simpler and less verbose due to reduced boilerplate code for communication and less control flow jumps. The claims (a), (c), and (d) are a direct consequence of Prisma’s design and do not require further evidence. Claim (c) has been formally proven in Section 3. It remains to investigate claim (e), i.e., in which extent Prisma reduces the amount of code and error-prone control-flow jumps.

To this end, we implemented all case studies with equivalent functionality in Prisma and in Solidity/JavaScript. The JavaScript client logic is in direct style using `async/await` – the Solidity contract needs to be implemented as a finite-state-machine. We keep the client logic of our case studies (in both, the Prisma and the Solidity implementation) as basic as possible, not to compare the client logic in Scala and in JavaScript but rather focus on the dApp semantics. A complex client logic would shadow the interaction with the contract logic – limited in size due to the gas semantics.

We start with comparing LOCs in the case studies (Figure 23). The results in Figure 23 show that case studies written in Prisma require only 55 – 89 % LOC compared to those implemented in Solidity/JavaScript. One exception is the standalone library, which has no client code and hence does not directly profit from the tierless design.

Second, we consider occurrences of explicit cross-tier control-flow calls in the Solidity/JavaScript dApps (cf. Table 22), which complicate control flow, compared to Prisma, where cross-tier access is seamless. In the client implementations, 6 – 18 % of all lines trigger a contract interaction passing the control flow to the contract and waiting for the control flow to return. From the contract code in finite-state-machine style, it is not directly apparent at which position the program flow continues, once passed back from clients to contract, i.e., which function is called by the clients next. Direct-style code, on the other hand, ensures that the control flow of the contract always continues in the line that passed the control flow to the client by invoking an `awaitCl` expression.

¹²A general solution is a much larger engineering effort and subject of industrial projects [42, 83]

$$\begin{aligned}
\text{comp}'(d; b; \text{trmp}(m)) &= d; \text{coclfn}(b, id, \text{assert}(\text{false}), \text{assert}(\text{false})); \text{trmp}(\text{comp}(m)) \\
&\quad \text{where } id \text{ fresh} \\
\text{comp} \left(\begin{array}{l} d; \text{coclfn}(b, id, \\ e_{1,alt}, \\ e_{2,alt}); \\ \text{tmp} \leftarrow_s ((\) \rightarrow e_1); e_2 \end{array} \right) &= \left(\begin{array}{l} d; \text{coclfn}(b, id, \\ \text{if let } (c :: \text{fv}(\) \rightarrow e_1) = id \text{ then } e_1 \text{ else } e'_{1,alt}, \\ \text{if let } (c :: x :: \text{fv}(x \rightarrow e'_2)) = id \text{ then} \\ \text{assert}(\text{this.state} == c \ \&\& \ \text{this.who}(\text{this.sender})); \\ \text{this.state} := 0; e'_2 \\ \text{else} \\ e'_{2,alt}); \\ \text{this.who} := e_0; \text{this.state} := c; \\ (\text{More}, c :: \text{fv}(\) \rightarrow e_1), c :: \text{fv}(x \rightarrow e'_2)) \end{array} \right) \\
&\quad \text{where } c \text{ fresh} \\
&\quad \text{and } d; \text{coclfn}(b, id, e'_{1,alt}, e'_{2,alt}); e'_2 = \\
&\quad \quad \text{defun}(d; \text{coclfn}(b, id, e_{1,alt}, e_{2,alt}); e_2) \\
\text{comp} \left(\begin{array}{l} d; \text{coclfn}(b, id, e_{1,alt}, e_{2,alt}); \\ x = e_0; e_1 \end{array} \right) &= \left(\begin{array}{l} d; \text{coclfn}(b, id, e_{1,alt}, e_{2,alt}); \\ x = e_0; \text{defun}(e_1) \end{array} \right) \\
\text{comp} \left(\begin{array}{l} d; \text{coclfn}(b, id, e_{1,alt}, e_{2,alt}); \\ e \end{array} \right) &= \left(\begin{array}{l} d; \text{coclfn}(b, id, e_{1,alt}, e_{2,alt}); \\ e \end{array} \right) \\
\text{coclfn}(b, id, e_{1,alt}, e_{2,alt}) &= (@\text{cl this. clfn} = id \rightarrow e_{1,alt}); (@\text{co this. cofn} = id \rightarrow e_{2,alt}); b
\end{aligned}$$

Fig. 24. comp' and comp .

$$\begin{aligned}
\text{fv}(m_0 :: m_1) &= \text{fv}(m_0) \cup \text{fv}(m_1) \\
\text{fv}(x \rightarrow m) &= \text{fv}(m) \setminus \text{fv}(x) \\
\text{fv}(id) &= \{id\} \\
\text{fv}(m_0 \ m_1) &= \text{fv}(m_0) \cup \text{fv}(m_1) \\
\text{fv}(\text{awaitCl}^*((m_0, (\) \rightarrow m_1)) &= \text{fv}(m_0) \cup \text{fv}(m_1) \\
\text{fv}(\text{let } x = m_0; m_1) &= \text{fv}(m_0) \cup \text{fv}(m_1) \setminus \text{fv}(x) \\
\text{fv}(\text{this.i} := m_0) &= \text{fv}(m_0) \\
\text{fv}(\text{this.j} := m_0) &= \text{fv}(m_0) \\
\text{fv}(\text{this.i}) &= \{\} \\
\text{fv}(\text{this.j}) &= \{\} \\
\text{fv}(c) &= \{\}
\end{aligned}$$

Fig. 25. Free variables.

C PROOFS

We provide the definition of comp' and comp in Figure 24, the definition for the free variables for a given term fv in Figure 25 and the detailed proofs for the theorem and the lemmas on the following pages.

THEOREM 1 (SECURE COMPILATION). For all programs P over closed terms, the trace set of evaluating the program under attack equals the trace set of evaluating the compiled program under attack, i.e.,

$$\forall P. \{ \text{init}_A(\text{comp}'(\text{mnf}'((P)))) \} \approx \dots \approx \{ \text{init}_A(P) \}$$

PROOF.

$$\begin{aligned} & \text{init}_A(\text{comp}'(\text{mnf}'(P))) \\ \stackrel{\text{Lemma 3}}{\approx} & \text{init}_A(\text{mnf}'(P)) \\ \stackrel{\text{Lemma 5}}{\approx} & \text{init}_A(P) \end{aligned}$$

□

Extensions. For simplicity, our definition of initialization uses a fixed set of clients. Yet, the malicious semantics does not actually depend on the fixed set of clients, but instead models an attacker that is in control of all clients with the capability of sending messages from any client, not bound to the fixed set. Hence, it is straightforward to extend the proofs to the setting of a dynamic set of clients, e.g., clients joining and leaving at run time.

Further, our trace equality relation defines that all programs in the relation eventually reduce to values, filtering out programs that loop or get stuck. Below, we outline an approach to prove trace equality for looping or stuck programs by showing that such programs loop with the same infinite trace or get stuck at the same trace, respectively. To this end, we track the number of steps done via a step-indexed trace equality relation:

$$p; q; e \Downarrow^n = \{ (p', v) \mid (p; q; e) \rightarrow^n (p'; q'; v) \} T \Downarrow^n = \bigcup_{p; q; e \in T} p; q; e \Downarrow^n$$

With this definition, we can no longer use just equality of traces as the left and right program may take a different number of steps to produce the same events. Instead, we move from an equality relation to a relation stating non-disagreement, which says that – independently of how long we run either statement – the traces will never be in disagreement:

$$(T \approx^n S) \Leftrightarrow (T \Downarrow^n \#_{\text{set}} S \Downarrow^n)$$

where $\#_{\text{set}}$ is defined on trace sets as

$$T \#_{\text{set}} S \Leftrightarrow (\forall t \in T. \exists s \in S. t \#_{\text{trace}} s) \wedge (\forall s \in S. \exists t \in T. t \#_{\text{trace}} s)$$

and $\#_{\text{trace}}$ on event traces as

$$\begin{aligned} (ev, ()) \#_{\text{trace}} (ev, \text{tail}_2) &= \text{true} \\ (ev, \text{tail}_1) \#_{\text{trace}} (ev, ()) &= \text{true} \\ (ev_1, \text{tail}_1) \#_{\text{trace}} (ev_2, \text{tail}_2) &= \text{false} \\ (ev, \text{tail}_1) \#_{\text{trace}} (ev, \text{tail}_2) &= \text{tail}_1 \#_{\text{trace}} \text{tail}_2 \end{aligned}$$

LEMMA 1 (ASSOC PRESERVES TRACES). *assoc* is defined as a recursive term-to-term transformation on open terms, whereas traceset equality is defined by reducing terms to values, i.e., on closed terms. Since all valid programs are closed terms, we show that *assoc* preserves the traceset of an open term e that is closed by substitution $[x \mapsto v]$.

For all terms e , traces p , traces q , values v , patterns x ,

$$\{ p; q; [x \mapsto v] \text{assoc}(e) \} \approx \dots \approx \{ p; q; [x \mapsto v] e \}$$

PROOF. By induction over term structure.

Case. $e = (\text{let } x_1 = (\text{let } x_0 = e_0; e_1); e_2)$.

We know $x_0 \notin \text{fv}(e_2)$ since e_2 is not in the scope of the x_0 binding, and that all identifiers are distinct, which can always be achieved by α -renaming.

$$x_0 \notin \text{fv}(e_2)$$

According to \approx , we only consider terms that reduce to a value. Therefore, let ϕ be the judgement that the term e_0 closed by $[x \mapsto v]$ with trace p evaluates to a value v_0 producing trace p_0 .

$$\phi \equiv (p; q; [x \mapsto v] e_0 \rightarrow^* p p_0; q; v_0)$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom (\rightarrow^*) and the original program from bottom to top (\leftarrow^*) until configurations converge. The induction hypothesis (IH) allows the removal of *assoc* in redex position under traceset equality (\approx).

$$\begin{array}{l}
\{ p; q; [x \mapsto v] \text{assoc}(e) \} \\
\stackrel{\text{def. } e}{=} \{ p; q; [x \mapsto v] \text{assoc}(\text{let } x_1 = (\text{let } x_0 = e_0; e_1); e_2) \} \\
\stackrel{\text{def. } \text{assoc}}{=} \{ p; q; [x \mapsto v] \text{assoc}(\text{let } x_0 = e_0; \text{assoc}(\text{let } x_1 = e_1; e_2)) \} \\
\stackrel{\text{IH}}{\approx} \{ p; q; [x \mapsto v] \text{let } x_0 = e_0; \text{assoc}(\text{let } x_1 = e_1; e_2) \} \\
\stackrel{\text{def. } \mapsto}{=} \{ p; q; \text{let } x_0 = [x \mapsto v] e_0; [x \mapsto v] \text{assoc}(\text{let } x_1 = e_1; e_2) \} \\
\stackrel{\phi}{\rightarrow^*} \{ p p_0; q; \text{let } x_0 = v_0; [x \mapsto v] \text{assoc}(\text{let } x_1 = e_1; e_2) \mid \forall v_0 p_0, \phi \} \\
\stackrel{\text{RLET}}{\rightarrow} \{ p p_0; q; [x_0 \mapsto v_0, x \mapsto v] \text{assoc}(\text{let } x_1 = e_1; e_2) \mid \forall v_0 p_0, \phi \} \\
\stackrel{\text{IH}}{\approx} \{ p p_0; q; [x_0 \mapsto v_0, x \mapsto v] \text{let } x_1 = e_1; e_2 \mid \forall v_0 p_0, \phi \} \\
\stackrel{\text{def. } \mapsto; x_0 \notin \text{fv}(e_2)}{=} \{ p p_0; q; \text{let } x_1 = [x_0 \mapsto v_0, x \mapsto v] e_1; [x \mapsto v] e_2 \mid \forall v_0 p_0, \phi \} \\
\stackrel{\text{RLET}}{\leftarrow} \{ p p_0; q; \text{let } x_1 = (\text{let } x_0 = v_0; [x \mapsto v] e_1); [x \mapsto v] e_2 \mid \forall v_0 p_0, \phi \} \\
\stackrel{\phi}{\leftarrow^*} \{ p; q; \text{let } x_1 = (\text{let } x_0 = [x \mapsto v] e_0; [x \mapsto v] e_1); [x \mapsto v] e_2 \} \\
\stackrel{\text{def. } \mapsto}{=} \{ p; q; [x \mapsto v] \text{let } x_1 = (\text{let } x_0 = e_0; e_1); e_2 \} \\
\stackrel{\text{def. } e}{=} \{ p; q; [x \mapsto v] e \}
\end{array}$$

Case. $e \neq (\text{let } x_1 = (\text{let } x_0 = e_0; e_1); e_2)$.

If e is not of nested let form, we simply apply the definition of *assoc*.

$$\begin{array}{l} \{ p; q; [x \Rightarrow v] \text{assoc}(e) \} \\ \text{def. } \stackrel{\text{assoc}}{=} \{ p; q; [x \Rightarrow v] e \} \end{array}$$

□

LEMMA 2 (MNF PRESERVES TRACES). *mnf* is defined as a recursive term-to-term transformation on open terms, whereas traceset equality is defined by reducing terms to values, i.e., on closed terms. Since all valid programs are closed terms, we show that *mnf* preserves the traceset of an open term e that is closed by substitution $[x \mapsto v]$.

For all terms e , traces p , traces q , values v , patterns x ,

$$\{ p; q; [x \mapsto v] \text{mnf}(e) \} \approx \dots \approx \{ p; q; [x \mapsto v] e \}$$

PROOF. By induction over term structure.

Case. $e = e_0 e_1$.

According to \approx , we only consider terms that reduce to a value. Therefore, let ϕ_0 be the judgement that the term e_0 closed by $[x \mapsto v]$ with trace p evaluates to a value v_0 producing trace p_0 . Let ϕ_1 be the judgement that the term e_1 closed by $[x \mapsto v]$ with trace $p p_0$ evaluates to a value v_1 producing trace $p p_0 p_1$.

$$\begin{aligned} \phi_0 &\equiv (p; q; [x \mapsto v] e_0 \rightarrow^* p p_0; q; v_0) \\ \phi_1 &\equiv (p p_0; q; [x \mapsto v] e_1 \rightarrow^* p p_0 p_1; q; v_1) \end{aligned}$$

Let id_0 be the fresh identifier *mnf* produces.

$$id_0 \text{ fresh}$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom (\rightarrow^*) and the original program from bottom to top (\leftarrow^*) until configurations converge. The induction hypothesis (IH) allows the removal of *mnf* in redex position under traceset equality (\approx).

$$\begin{aligned} & \{ p; q; [x \mapsto v] \text{mnf}(e) \} \\ \stackrel{\text{def. } e}{=} & \{ p; q; [x \mapsto v] \text{mnf}(e_0 e_1) \} \\ \stackrel{\text{def. } \text{mnf}}{=} & \{ p; q; [x \mapsto v] \text{assoc}(\text{let } id_0 = \text{mnf}(e_0); \text{assoc}(\text{let } id_1 = \text{mnf}(e_1); id_0 id_1)) \} \\ \stackrel{\text{Lemma 1}}{\approx} & \{ p; q; [x \mapsto v] \text{let } id_0 = \text{mnf}(e_0); \text{assoc}(\text{let } id_1 = \text{mnf}(e_1); id_0 id_1) \} \\ \stackrel{\text{def. } \mapsto}{=} & \{ p; q; \text{let } id_0 = [x \mapsto v] \text{mnf}(e_0); [x \mapsto v] \text{assoc}(\text{let } id_1 = \text{mnf}(e_1); id_0 id_1) \} \\ \stackrel{IH}{\approx} & \{ p; q; \text{let } id_0 = [x \mapsto v] e_0; [x \mapsto v] \text{assoc}(\text{let } id_1 = \text{mnf}(e_1); id_0 id_1) \} \\ \stackrel{\phi_0}{\rightarrow^*} & \{ p p_0; q; \text{let } id_0 = v_0; [x \mapsto v] \text{assoc}(\text{let } id_1 = \text{mnf}(e_1); id_0 id_1) \mid \forall v_0 p_0, \text{ if } \phi_0 \} \\ \stackrel{\text{RLET}}{\rightarrow} & \{ p p_0; q; [id_0 \mapsto v_0, x \mapsto v] \text{assoc}(\text{let } id_1 = \text{mnf}(e_1); id_0 id_1) \mid \forall v_0 p_0, \text{ if } \phi_0 \} \\ \stackrel{\text{Lemma 1}}{\approx} & \{ p p_0; q; [id_0 \mapsto v_0, x \mapsto v] \text{let } id_1 = \text{mnf}(e_1); id_0 id_1 \mid \forall v_0 p_0, \text{ if } \phi_0 \} \\ \stackrel{\text{def. } \mapsto}{=} & \{ p p_0; q; \text{let } id_1 = [id_0 \mapsto v_0, x \mapsto v] \text{mnf}(e_1); v_0 id_1 \mid \forall v_0 p_0, \text{ if } \phi_0 \} \\ \stackrel{IH}{=} & \{ p p_0; q; \text{let } id_1 = [id_0 \mapsto v_0, x \mapsto v] e_1; v_0 id_1 \mid \forall v_0 p_0, \text{ if } \phi_0 \} \\ id_0 \stackrel{\text{fresh}}{=} & \{ p p_0; q; \text{let } id_1 = [x \mapsto v] e_1; v_0 id_1 \mid \forall v_0 p_0, \text{ if } \phi_0 \} \\ \stackrel{\phi_1}{\rightarrow^*} & \{ p p_0 p_1; q; \text{let } id_1 = v_1; v_0 id_1 \mid \forall v_0 v_1 p_0 p_1, \text{ if } \phi_0, \phi_1 \} \end{aligned}$$

$$\begin{array}{l}
\text{RLET} \rightarrow \quad \{ p \ p_0 \ p_1; q; v_0 \ v_1 \mid \forall v_0 \ v_1 \ p_0 \ p_1, \text{ if } \phi_0, \phi_1 \} \\
\phi_1 \leftarrow^* \quad \{ p \ p_0; q; v_0 \ [x \mapsto v] \ e_1 \mid \forall v_0 \ p_0, \text{ if } \phi_0 \} \\
\phi_0 \leftarrow^* \quad \{ p; q; ([x \mapsto v] \ e_0) \ [x \mapsto v] \ e_1 \} \\
\text{def.} \models \quad \{ p; q; [x \mapsto v] \ e_0 \ e_1 \} \\
\text{def.} \equiv e \quad \{ p; q; [x \mapsto v] \ e \}
\end{array}$$

Case. $e = \text{let } id_0 = e_0; e_1$.

According to \approx , we only consider terms that reduce to a value. Therefore, let ϕ be the judgement that the term e_0 closed by $[x \mapsto v]$ with trace p evaluates to a value v_0 producing trace p_0 .

$$\phi_0 \equiv (p; q; [x \mapsto v] \ e_0 \rightarrow^* p \ p_0; q; v_0)$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom (\rightarrow^*) and the original program from bottom to top (\leftarrow^*) until configurations converge. The induction hypothesis (IH) allows the removal of mnf in redex position under traceset equality (\approx).

$$\begin{array}{l}
\{ p; q; [x \mapsto v] \ mnf(e) \} \\
\text{def.} \equiv e \quad \{ p; q; [x \mapsto v] \ mnf(\text{let } id_0 = v_0; e_1) \} \\
\text{def.} \equiv mnf \quad \{ p; q; [x \mapsto v] \ assoc(\text{let } id_0 = mnf(e_0); mnf(e_1)) \} \\
\text{Lemma 1} \quad \approx \quad \{ p; q; [x \mapsto v] \ \text{let } id_0 = mnf(e_0); mnf(e_1) \} \\
\text{def.} \models \quad \{ p; q; \text{let } id_0 = [x \mapsto v] \ mnf(e_0); [x \mapsto v] \ mnf(e_1) \} \\
\text{IH} \quad \approx \quad \{ p; q; \text{let } id_0 = [x \mapsto v] \ e_0; [x \mapsto v] \ mnf(e_1) \} \\
\phi_0 \rightarrow^* \quad \{ p \ p_0; q; \text{let } id_0 = v_0; [x \mapsto v] \ mnf(e_1) \mid \forall v_0 \ p_0, \text{ if } \phi_0 \} \\
\text{RLET} \rightarrow \quad \{ p \ p_0; q; [id_0 \mapsto v_0, x \mapsto v] \ mnf(e_1) \mid \forall v_0 \ p_0, \text{ if } \phi_0 \} \\
\text{IH} \quad \approx \quad \{ p \ p_0; q; [id_0 \mapsto v_0, x \mapsto v] \ e_1 \mid \forall v_0 \ p_0, \text{ if } \phi_0 \} \\
\text{RLET} \leftarrow \quad \{ p \ p_0; q; [x \mapsto v] \ \text{let } id_0 = v_0; e_1 \mid \forall v_0 \ p_0, \text{ if } \phi_0 \} \\
\phi_0 \leftarrow^* \quad \{ p; q; [x \mapsto v] \ \text{let } id_0 = e_0; e_1 \} \\
\text{def.} \equiv e \quad \{ p; q; [x \mapsto v] \ e \}
\end{array}$$

Case. The other cases of e are proved analogously. □

LEMMA 3 (MNF' PRESERVES TRACE). mnf' is defined on programs. To evaluate a program, it is initialized with a set of clients A . mnf' preserves the traceset of (closed) programs P for any set of clients A .

For all P ,

$$\{ \text{init}_A(mnf'(P)) \} \approx \dots \approx \{ \text{init}_A(P) \}$$

PROOF. By induction over term structure.

Case. $P = (d; b; e_0)$.

Initializing the definitions $d; b$ with A produces the trace p and the state q .

$$\text{init}_A(d; b) = p; q$$

According to \approx , we only consider terms that reduce to a value. Therefore, let ϕ be the judgement that the term e_0 closed by $[x \mapsto v]$ in trace p produces a value v_0 and trace p_0 .

$$\phi \equiv (p; q; e_0 \rightarrow^* p_0; q; v_0)$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom (\rightarrow^*) and the original program from bottom to top (\leftarrow^*) until configurations converge, using Lemma 2.

$$\begin{array}{l}
\{ \text{init}_A(mnf'(P)) \} \\
\stackrel{\text{def. } P}{=} \{ \text{init}_A(mnf'(d; b; e_0)) \} \\
\stackrel{\text{def. } mnf'}{=} \{ \text{init}_A(d; b; \text{trmp}(mnfe(\text{Done}(e_0)))) \} \\
\stackrel{\text{def. } \text{init}_A}{=} \{ p; q; \text{trmp}(mnfe(\text{Done}(e_0))) \} \\
\stackrel{\text{Lemma 2}}{\approx} \{ p; q; \text{trmp}(\text{Done}(e_0)) \} \\
\stackrel{\phi}{\rightarrow^*} \{ p_0; q; \text{trmp}(\text{Done}(v_0)) \mid \forall v_0 p_0, \text{ if } \phi \} \\
\stackrel{\text{RDONE}}{\rightarrow} \{ p_0; q; v_0 \mid \forall v_0 p_0, \text{ if } \phi \} \\
\stackrel{\phi}{\leftarrow^*} \{ p; q; e_0 \} \\
\stackrel{\text{def. } \text{init}_A}{=} \{ \text{init}_A(d; b; e_0) \} \\
\stackrel{\text{def. } P}{=} \{ \text{init}_A(P) \}
\end{array}$$

□

LEMMA 4 (COMP PRESERVES TRACES). *comp* is defined on programs. To evaluate a program, it is initialized with a set of clients A . *comp* preserves the traceset of (closed) programs P for any set of clients A .

For all definitions b , definitions d , terms e , values v , patterns x ,

$$\{ [x \Rightarrow v] \text{init}_A(\text{comp}(d; b; \text{trmp}(e))) \} \approx \dots \approx \{ \text{init}_A(d; b; \text{trmp}([x \Rightarrow v] e)) \}$$

PROOF. By induction over term structure.

Case. $e = \text{let } x = \text{awaitCl}_s((e_0, () \rightarrow e_1)); e_2$.

comp expects the definitions b to be of form:

$$b = \begin{pmatrix} @cl \text{ this.cfn} = id \rightarrow e_{1,alt}; \\ @co \text{ this.cofn} = id \rightarrow e_{2,alt}; \\ b_{rest} \end{pmatrix}$$

comp is defined recursively and applied to the term e_2 . Intuitively, *comp* transforms e_2 to e'_2 and b to b' by moving the part of e_2 that comes after the awaitCl_s call into the cofn definition inside b . The recursive call is given as follows:

$$(d; b'; \text{trmp}(e'_2)) = \text{comp}(d; b; \text{trmp}(e_2))$$

$$b' = \begin{pmatrix} @cl \text{ this.cfn} = id \rightarrow e'_{1,alt}; \\ @co \text{ this.cofn} = id \rightarrow e'_{2,alt}; \\ b_{rest} \end{pmatrix}$$

After the recursive call, *comp* moves the transformed e'_2 into the cofn definition, resulting in e' and b'' with $e''_{1,alt}$ and $e''_{2,alt}$.

$$\phi \equiv (\{ d; b''; \text{trmp}(e') \} = \{ \text{comp}(d; b; \text{trmp}(e)) \})$$

$$b'' = \begin{pmatrix} @cl \text{ this.cfn} = id \rightarrow e''_{1,alt}; \\ @co \text{ this.cofn} = id \rightarrow e''_{2,alt}; \\ b_{rest} \end{pmatrix}$$

$$e''_{1,alt} = \begin{pmatrix} \text{if let } (c :: \text{fv}(() \rightarrow e_1)) = id \\ \text{then } e_1 \\ \text{else } e'_{2,alt} \end{pmatrix}$$

$$e''_{2,alt} = \begin{pmatrix} \text{if let } (c :: x :: \text{fv}(x \rightarrow e'_2)) = id \\ \text{then assert}(\text{this.state} == c \ \&\& \ \text{this.sender} == \text{this.who}); \\ \text{this.state} := 0; e'_2 \\ \text{else } e'_{2,alt} \end{pmatrix}$$

Let $p; q$ be the trace and state produced by initializing $d; b$ with A , and $p; q'$ for initializing $d; b'$, and $p; q''$ for initializing $d; b''$.

$$\begin{aligned} \text{init}_A(d; b) &= p; q \\ \text{init}_A(d; b') &= p; q' \\ \text{init}_A(d; b'') &= p; q'' \end{aligned}$$

According to \approx , we only consider terms that reduce to a value. Therefore, let ϕ_0 be the judgement that the term e_0 closed by $[x \Rightarrow v]$ in trace p produces a value v_0 and trace p_1 .

$$\phi_0(q_\phi) = (p; q_\phi; [x \Rightarrow v] e_0 \rightarrow p \ p_1; q_\phi; v_0)$$

We define ϕ_1 based on ϕ :

$$\begin{aligned}
& \phi \\
& = \\
& \{ d; b''; \text{trmp}(e') \} = \{ \text{comp}(d; b; \text{trmp}(e)) \} \\
& \quad \rightarrow \text{generalize } [x \mapsto v] \text{init}_A(\dots) \\
& \{ [x \mapsto v] \text{init}_A(d; b''; \text{trmp}(e')) \} = \{ [x \mapsto v] \text{init}_A(\text{comp}(d; b; \text{trmp}(e))) \} \\
& \quad \rightarrow (= \rightarrow \approx) \\
& \{ [x \mapsto v] \text{init}_A(d; b''; \text{trmp}(e')) \} \approx \{ [x \mapsto v] \text{init}_A(\text{comp}(d; b; \text{trmp}(e))) \} \\
& \quad \rightarrow IH \\
& \{ [x \mapsto v] \text{init}_A(d; b''; \text{trmp}(e')) \} \approx \{ \text{init}_A(d; b; \text{trmp}([x \mapsto v] e)) \} \\
& \quad \rightarrow \text{def. init}_A \\
& \{ p; q''; \text{trmp}([x \mapsto v] e') \} \approx \{ p; q; \text{trmp}([x \mapsto v] e) \} \\
& \quad \equiv \\
& \phi_1
\end{aligned}$$

The lemma holds by the following chain of transitive relations. We evaluate the compiled program from top to bottom (\rightarrow^*) and the original program from bottom to top (\leftarrow^*) until configurations converge.

$$\begin{array}{l}
\{ [x \mapsto v] \text{init}_A(\text{comp}(d; b; \text{trmp}(e))) \} \\
\text{def. } \stackrel{e}{=} \{ [x \mapsto v] \text{init}_A(\text{comp}(d; b; \text{trmp}(\text{let } x_3 = \text{awaitCl}_s((e_0, () \rightarrow e_1)); e_2))) \} \\
\text{def. } \stackrel{\text{comp}}{=} \left\{ \begin{array}{l} [x \mapsto v] \text{init}_A(d; b''; \text{trmp}(\\ \text{this.who} := e_0; \text{this.state} := c; \\ \text{More}(c :: \text{fv}(() \rightarrow e_1), c :: \text{fv}(x \rightarrow e'_2))) \end{array} \right\} \\
\text{def. } \stackrel{\text{init}_A}{=} \left\{ \begin{array}{l} p; q''; [x \mapsto v] \text{trmp}(\\ \text{this.who} := e_0; \text{this.state} := c; \\ \text{More}(c :: \text{fv}(() \rightarrow e_1), c :: \text{fv}(x \rightarrow e'_2)) \end{array} \right\} \\
\text{def. } \stackrel{\mapsto}{=} \left\{ \begin{array}{l} p; q''; \text{trmp}(\\ \text{this.who} := [x \mapsto v] e_0; \text{this.state} := c; \\ \text{More}(c :: [x \mapsto v] \text{fv}(() \rightarrow e_1), c :: [x \mapsto v] \text{fv}(x \rightarrow e'_2)) \end{array} \right\} \\
\text{def. } \stackrel{\phi_0(q'')}{\rightarrow^*} \left\{ \begin{array}{l} p \ p_1; q''; \text{trmp}(\\ \text{this.who} := v_0; \text{this.state} := c; \\ \text{More}(c :: [x \mapsto v] \text{fv}(() \rightarrow e_1), c :: [x \mapsto v] \text{fv}(x \rightarrow e'_2)) \\ | \forall v_0 \ p_1, \text{ if } \phi_0 \end{array} \right\} \\
\text{RSET}^\dagger, \text{RSET}^\dagger \rightarrow \left\{ \begin{array}{l} p \ p_1; q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto c]; \\ \text{trmp}(\text{More}(c :: [x \mapsto v] \text{fv}(() \rightarrow e_1), c :: [x \mapsto v] \text{fv}(x \rightarrow e'_2)) \\ | \forall v_0 \ p_1, \text{ if } \phi_0 \end{array} \right\} \\
\text{RMORE} \rightarrow \left\{ \begin{array}{l} p \ p_1; q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto c]; \\ \text{tmp} \leftarrow_t \text{this.cfn}(c :: [x \mapsto v] \text{fv}(() \rightarrow e_1)); \\ \text{trmp}(\text{this.cfn}(c :: \text{tmp} :: [x \mapsto v] \text{fv}(x \rightarrow e'_2))) \\ | \forall v_0 \ p_1, \text{ if } \phi_0 \end{array} \right\}
\end{array}$$

$\xrightarrow{\text{RBT}}$	$\left\{ \begin{array}{l} p \ p_1 \ \text{msg}(v'_0, v'_2) \ \text{wr}(0, \text{sender}, v'_0); \ q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto c]; \\ \text{let } tmp = v'_2; \ \text{trmp}(\text{this.cofn}(c :: tmp :: [x \mapsto v] \ \text{fv}(x \rightarrow e'_2))) \\ \ \forall v_0 \ p_1 \ v'_0 \ v'_2, \ \text{if } \phi_0 \end{array} \right\}$
$\stackrel{\text{case } v'_0 = v_0}{=}$	$\left\{ \begin{array}{l} p \ p_1 \ \text{msg}(v'_0, v'_2) \ \text{wr}(0, \text{sender}, v'_0); \ q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto c]; \\ \text{let } tmp = v'_2; \ \text{trmp}(\text{this.cofn}(c :: tmp :: [x \mapsto v] \ \text{fv}(x \rightarrow e'_2))) \\ \ \forall v_0 \ p_1 \ v'_0 \ v'_2, \ \text{if } v'_0 \neq v_0, \ \phi_0 \\ \hline p \ p_1 \ \text{msg}(v'_0, v'_2) \ \text{wr}(0, \text{sender}, v'_0); \ q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto c]; \\ \text{let } tmp = v'_2; \ \text{trmp}(\text{this.cofn}(c :: tmp :: [x \mapsto v] \ \text{fv}(x \rightarrow e'_2))) \\ \ \forall v_0 \ p_1 \ v'_0 \ v'_2, \ \text{if } v'_0 = v_0, \ \phi_0 \end{array} \right\}$
$\xrightarrow{*}$ RLET, RGET, RAPP, RT, RGET, ROP, RGET, RGET, ROP, ROP	$\left\{ \begin{array}{l} p \ p_1 \ \text{msg}(v'_0, v'_2) \ \text{wr}(0, \text{sender}, v'_0); \ q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto c]; \\ \text{trmp}(\text{assert}(\text{false}); \ \text{this.state} := 0; \ [x \mapsto v'_2, \ x \mapsto v] \ e'_2) \\ \ \forall v_0 \ p_1 \ v'_0 \ v'_2, \ \text{if } v'_0 \neq v_0, \ \phi_0 \\ \hline p \ p_1 \ \text{msg}(v'_0, v'_2) \ \text{wr}(0, \text{sender}, v'_0); \ q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto c]; \\ \text{trmp}(\text{assert}(\text{true}); \ \text{this.state} := 0; \ [x \mapsto v'_2, \ x \mapsto v] \ e'_2) \\ \ \forall v_0 \ p_1 \ v'_0 \ v'_2, \ \text{if } v'_0 = v_0, \ \phi_0 \end{array} \right\}$
\approx def.	$\left\{ \begin{array}{l} p \ p_1 \ \text{msg}(v'_0, v'_2) \ \text{wr}(0, \text{sender}, v'_0); \ q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto c]; \\ \text{trmp}(\text{assert}(\text{true}); \ \text{this.state} := 0; \ [x \mapsto v'_2, \ x \mapsto v] \ e'_2) \\ \ \forall v_0 \ p_1 \ v'_0 \ v'_2, \ \text{if } v'_0 = v_0, \ \phi_0 \end{array} \right\}$
$\xrightarrow{*}$ RLET, RSET	$\left\{ \begin{array}{l} p \ p_1 \ \text{msg}(v'_0, v'_2) \ \text{wr}(0, \text{sender}, v'_0); \ q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto 0]; \\ \text{trmp}([x \mapsto v'_2, \ x \mapsto v] \ e'_2) \\ \ \forall v_0 \ p_1 \ v'_0 \ v'_2, \ \text{if } v'_0 = v_0, \ \phi_0 \end{array} \right\}$
$\stackrel{v'_0 = v_0}{=}$	$\left\{ \begin{array}{l} p \ p_1 \ \text{msg}(v_0, v'_2) \ \text{wr}(0, \text{sender}, v_0); \ q'' \ [\text{who} \mapsto v_0, \text{state} \mapsto 0]; \\ \text{trmp}([x \mapsto v'_2, \ x \mapsto v] \ e'_2) \\ \ \forall v_0 \ p_1 \ v'_2, \ \text{if } \phi_0 \end{array} \right\}$
\approx ϕ_1	$\left\{ \begin{array}{l} p \ p_1 \ \text{msg}(v_0, v'_2) \ \text{wr}(0, \text{sender}, v_0); \ q; \\ \text{trmp}([x \mapsto v'_2, \ x \mapsto v] \ e_2) \\ \ \forall v_0 \ p_1 \ v'_2, \ \text{if } \phi_0 \end{array} \right\}$
$\xleftarrow{*}$ RLET, RBS	$\left\{ \begin{array}{l} p \ p_1; \ q; \ \text{trmp}(\text{let } x = \text{awaitCl}_s(v_0, () \rightarrow [x \mapsto v] \ e_1); \ [x \mapsto v] \ e_2) \\ \ \forall v_0 \ p_1, \ \text{if } \phi_0 \end{array} \right\}$
$\xleftarrow{*}$ $\phi_0(q)$	$\left\{ p; \ q; \ \text{trmp}(\text{let } x = \text{awaitCl}_s([x \mapsto v] \ e_0, () \rightarrow [x \mapsto v] \ e_1); \ [x \mapsto v] \ e_2) \right\}$
$\stackrel{\text{def. } \Rightarrow}{=}$	$\left\{ p; \ q; \ \text{trmp}([x \mapsto v] \ \text{let } x = \text{awaitCl}_s(e_0, () \rightarrow e_1); \ e_2) \right\}$
$\stackrel{\text{def. } e}{=}$	$\left\{ p; \ q; \ \text{trmp}([x \mapsto v] \ e) \right\}$
$\stackrel{\text{def. } \text{init}_A}{=}$	$\left\{ \text{init}_A(b; \ d; \ \text{trmp}([x \mapsto v] \ e)) \right\}$

Case. $e = x_0$.

Let $p; q$ be the trace and state produced by initializing $d; b$ with A .

$$\text{init}_A(d; b) = p; q$$

The traceset equality holds by definition of comp and init_A .

$$\stackrel{\text{def. } e}{=} \left\{ [x \mapsto v] \ \text{init}_A(\text{comp}(d; b; \ \text{trmp}(e))) \right\}$$

$$\left\{ [x \mapsto v] \ \text{init}_A(\text{comp}(d; b; \ \text{trmp}(x_0))) \right\}$$

$$\begin{aligned}
\text{def. } \underline{=}^{comp} & \{ [x \Rightarrow v] \text{init}_A(d; b; \text{comp}(\text{trmp}(x_0))) \} \\
\text{def. } \underline{=}^{init_A} & \{ p; q; [x \Rightarrow v] \text{trmp}(x_0) \} \\
\text{def. } \underline{=}^{\Rightarrow} & \{ p; q; \text{trmp}([x \Rightarrow v] x_0) \} \\
\text{def. } \underline{=}^e & \{ p; q; \text{trmp}([x \Rightarrow v] e) \} \\
\text{def. } \underline{=}^{init_A} & \{ \text{init}_A(d; b; \text{trmp}([x \Rightarrow v] e)) \}
\end{aligned}$$

□

LEMMA 5 (COMP' PRESERVES TRACES). $comp'$ is defined on programs. To evaluate a program, it is initialized with a set of clients A . $comp'$ preserves the traceset of (closed) programs P for any set of clients A .

For all definitions b , definitions d , terms e_0 ,

$$\{ \text{init}_A(\text{comp}'(d; b; \text{trmp}(e_0))) \} \approx \dots \approx \{ \text{init}_A(d; b'; \text{trmp}(e_0)) \}$$

PROOF. By induction over term structure.

Case. $P = (d; b; e_0)$.

Intuitively, $comp'$ prepends the definitions b with initial definitions for clfn and cofn that only contain $\text{assert}(\text{false})$, such that $comp$ can be applied.

$$b' = \left(\begin{array}{l} @\text{cl this.cfn} = id \rightarrow \text{assert}(\text{false}); \\ @\text{co this.cofn} = id \rightarrow \text{assert}(\text{false}); \\ b \end{array} \right)$$

The lemma holds by definition of $comp'$, and Lemma 4.

$$\begin{array}{l} \{ \text{init}_A(\text{comp}'(d; b; \text{trmp}(e_0))) \} \\ \stackrel{\text{def. } comp'}{=} \{ \text{init}_A(\text{comp}(d; b'; \text{trmp}(e_0))) \} \\ \stackrel{\text{Lemma 4}}{\approx} \{ \text{init}_A(d; b'; \text{trmp}(e_0)) \} \end{array}$$

□