TECHNISCHE UNIVERSITÄT DARMSTADT

Improving Scalability, Privacy, and Decentralization of Blockchains and their Applications via Multiparty Computation

Vom Fachbereich Informatik der TU Darmstadt genehmigte

Dissertation

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

von

Benjamin Schlosser (M.Sc.)

Darmstadt 2024

Gutachter:	Prof. Sebastian Faust, Ph.D.
	Prof. Carmit Hazay, Ph.D.
Datum der Einreichung:	15.07.2024

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.

(B. Schlosser)

Wissenschaftlicher Werdegang

Oktober 2014 - September 2017: Bachelor of Science in Informatik an der Technischen Universität Darmstadt.

Oktober 2017 - September 2019 : Master of Science in IT-Sicherheit an der Technischen Universität Darmstadt.

Oktober 2019 - August 2024: Doktorand am Fachgebiet für Angewandte Kryptographie an der Technischen Universität Darmstadt bei Prof. Sebastian Faust.

Acknowledgments

The last few years have been a very adventurous and instructive time. While working on a PhD is always connected with challenging periods, the pleasing times outweighed them by far for me. During this time, I learned a lot as a scientist and even more as a person. I was lucky to work with many expert and friendly colleagues and collaborators, which I would like to thank.

First, I would like to thank my supervisor, Sebastian Faust. After writing my Master's thesis in his group, he provided me the opportunity to start this journey. On this path, he offered to me to visit conferences and travel to lovely places. In the everyday life of a member of his group, he created a nice and friendly surrounding, which was the foundation for an enjoyable working atmosphere. Additionally, he connected me to outstanding cryptographers in many projects. One particular person who greatly impacted my PhD time was Carmit Hazay. Carmit was part of many projects from the start of my journey. She was part of countless discussions and provided priceless feedback. She even welcomed me in Tel Aviv for a research visit. I learned a lot about science and about myself as a person from her advice. I am very grateful for the mentorship she provided and for being a part of my disputation committee.

I would also like to thank Ahmad-Reza Sadeghi and Zsolt István for agreeing to be part of my disputation committee and Dorothee Nikolaus for helping me with the administrative work when submitting my thesis and before.

One particular reason that led to my decision to start my PhD studies was the extremely friendly and supportive working atmosphere in the CAC group. Therefore, I thank all my colleagues for a lovely time in and outside the office, joint conference visits, travels in foreign countries, and an enjoyable retreat in Sardinia - a special thanks to my colleague David, who shared an office with me and worked with me on many projects.

Finally, I would like to thank my parents, siblings, and wife. They have all created a stable family environment for me and supported me in all areas of life. My parents encouraged me my entire life to follow my own path and supported me without any doubt.

List of Own Publications

Peer-reviewed Publications

[4] H. Amler, L. Eckey, S. Faust, M. Kaiser, P. G. Sandner, and B. Schlosser. “DeFining DeFi: Challenges & Pathway”. In: *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services, BRAINS 2021, Paris, France, September 27-30, 2021.* 2021, pp. 181–184.

[91] L. Eckey, S. Faust, and B. Schlosser. “OptiSwap: Fast Optimistic Fair Exchange”. In: *ASIA CCS ’20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020.* 2020, pp. 543–557.

[94] 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**.

[95] 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**.

[96] 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**.

[97] 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.

[99] 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**.

[145] M. Lohr, B. Schlosser, J. Jürjens, and S. Staab. “Cost Fairness for Blockchain-Based Two-Party Exchange Protocols”. In: *IEEE International Conference on Blockchain, Blockchain 2020, Rhodes, Greece, November 2-6, 2020.* 2020, pp. 428–435.

Articles in Submission

[93] S. Faust, C. Hazay, D. Kretzler, L. Rometsch, and B. Schlosser. *Non-Interactive Threshold BBS+ From Pseudorandom Correlations*. Cryptology ePrint Archive, Paper 2023/1076. `https://eprint.iacr.org/2023/1076`. 2023. **Part of this thesis**.

My Contribution

The results of this thesis are based on five research papers. The papers resulted from collaborations between myself and several exceptional researchers: Sebastian Faust (TU Darmstadt), Tommaso Frassetto (TU Darmstadt), Carmit Hazay (Bar-Ilan University), Patrick Jauernig (TU Darmstadt), David Koisser (TU Darmstadt), David Kretzler (TU Darmstadt), Leandro Rometsch (TU Darmstadt), and Ahmad-Reza Sadeghi (TU Darmstadt). I thank all of them for the fruitful collaborations. Next, I will detail my contribution in all five publications.

Chapter 3 is based on the publication [99]. Together with Tommaso Frassetto, Patrick Jauernig, David Koisser, and David Kretzer, I was involved in all discussions leading to the design and modeling of our off-chain protocol in [99]. Subsequently, David Kretzler and I jointly designed and specified the protocol. Additionally, David Kretzler and I jointly proved our protocol's security properties, i.e., correctness, liveness, and state privacy.

Chapter 4 is based on the publications [94, 95, 96]. All three publications resulted from close discussions between Sebastian Faust, Carmit Hazay, David Kretzler, and myself. In [96], we proposed a covertly secure composition of an offline and online phase based on the TinyOT protocol. To this end, I initially observed the TinyOT protocol provides better communication complexity when reducing the MAC length in the online phase. Next, David and I jointly defined a composition theorem stating that the combination of covertly secure offline and online phases is also covertly secure. In this step, I focused on the formalization and the proof of the theorem, compared our theorem with previous composition theorems, and provided a discussion about constraints, which David identified. For designing a covertly secure offline phase, I compared the communication complexity of a cut-and-choose-based approach to a covertly secure variant of the TinyOT's precomputation based on reducing the statistical security parameter. As a result of this comparison, I identified the former to provide better communication complexity for reasonable deterrence factors. In [95], David and I jointly designed and specified our protocols, and I supported David in conducting the security proof. Additionally, I defined the notion of a verifiable time-lock puzzle and specified a construction satisfying the new notion. Finally, I evaluated the efficiency of our protocols. In [94], I modeled the blockchain and judge functionalities and formulated the notion of financially backed covert security. The definition of our new notion included two new security properties for financial accountability and financial defamation freeness. Moreover, I supported David in presenting our protocols.

Chapter 5 is based on the publication [93], which resulted from collaboration with Sebastian Faust, Carmit Hazay, David Kretzler, and Leandro Rometsch. In [93], I contributed the design, specification, and security proof of our online phase. Additionally, I formalized the notion of strong reusable pseudorandom correlation functions (rPCFs) and proved that existing constructions satisfy this notion. Moreover, I designed and proved the security of our rPCF-based precomputation protocol.

Abstract

Since the advent of Bitcoin in 2008, a myriad of blockchain systems have emerged. Blockchains provide decentralized systems aiming to remove any trust in centralized parties. While Bitcoin provides simple money transfer and rudimentary scripting capabilities, other blockchains like Ethereum support the execution of complex smart contracts. Smart contrast sparked the invention of many new applications over blockchains, with decentralized finance (DeFi) being one of the most prominent. By moving financial services and products to decentralized and open blockchains, DeFi has the potential to democratize the financial market. While showing a promising feature, state-of-the-art blockchains still suffer from limitations and open problems. Limited scalability prevents mass adaption since the number of tolerable actions within the system is too low. Additionally, many systems lack strong privacy features, preventing their applicability to applications with high privacy requirements, like in the healthcare sector. Despite these open problems, blockchains are used in more and more new contexts due to their attractive features based on their decentralized nature. One example is the concept of self-sovereign identities (SSI), where blockchains provide decentralized storage of public metadata. In many new contexts, blockchains are paired with additional components, often not explicitly designed for blockchain applications. Hence, it remains an open problem to align these components with the fundamental idea of blockchains, i.e., removing trust in centralized parties.

In this thesis, we significantly contribute to the design of new solutions to all three mentioned problems. More concretely, we tackle the scalability and privacy problem and mitigate the trust in centralized parties in a new component combined with blockchains. Our main building block in all our contributions is secure multiparty computation (MPC), which allows distrusting parties to compute on private data without leaking anything except the output of the computation. First, we present a new off-chain protocol that supports the execution of smart contracts. Since prior work suffers from different shortcomings, our solution addresses them all simultaneously. Second, we use MPC to facilitate private computation for blockchains. To do so, we consider a security model that provides a trade-off between efficiency and security. For this setting, we propose further efficiency improvements, present a compiler for enhancing security, and propose a protocol to combine MPC with blockchains. Our final result allows parties to perform computation privately, and the computation's result defines a distribution of coins. Third, we look at anonymous credentials, an essential component of self-sovereign identities. We present a distributed issuance protocol for anonymous credentials based on the BBS+ signature scheme.

Contents

1. Introduction

In 2008, Bitcoin [158], the first decentralized cryptocurrency, was introduced, paving the way for a new payment paradigm. As of today, many payment systems rely on trusted parties like banks or companies (e.g., PayPal) to execute client transactions. In contrast, the fundamental idea of decentralized cryptocurrencies is to replace trusted third parties with a decentralized system. These systems benefit, among other things, from eliminating the risk of a single point of failure and providing censorship resistance, as no single party can block transactions.

The state of the Bitcoin system, i.e., the parties' balances, is stored in a so-called blockchain. A blockchain is a data structure where transactions are packed into blocks, and each block is linked to the previous block through a cryptographic hash function. This results in a chain of blocks, therefore called a blockchain.

The blockchain is maintained by a set of parties instead of a centralized party. More concretely, parties wishing to transfer money send a transaction to the network. The network nodes, called miners, pack several transactions into a block and link the new block to the previous block. The miners use a consensus protocol to agree on new blocks. Such a protocol ensures that all parties have a common state of the blockchain. A common view of the state allows every miner to compute all parties' balances and prevent double-spending attacks. The Bitcoin system builds on the Nakamoto consensus algorithm [158], which uses a proof-of-work (PoW) algorithm [87] to determine which miners are eligible to add the next block. Specifically, miners must solve a computationally hard puzzle to add a new block to the chain. A PoW-based consensus algorithm guarantees the system's correctness as long as the majority of computing power is controlled by honest parties, shifting away from trusting a single party towards trusting an honest majority.

Since 2008, a myriad of blockchain systems have emerged [137]. Each aims to improve on one or several aspects compared to prior systems. For instance, they increase the throughput, use a different consensus algorithm to waste less energy than PoW, or support more applications (e.g., [57, 92, 174, 202, 203]). The most popular are Bitcoin and Ethereum [203]. Both are permissionless systems, where parties may join and leave the system anytime. In contrast, permissioned systems allow only permitted parties to join the system. In this thesis, we focus on the permissionless setting.

While Bitcoin supports money transfer and minimal scripting capabilities, other blockchain systems like Ethereum allow the execution of Turing-complete programs. These programs are called smart contracts and open the opportunity for many new applications. Nowadays, not only is simple money transfer possible over a blockchain system, but also playing games like CryptoKitties [71], realizing fair exchanges [15, 88, 91], and much more. One application that remarkably increased the interest in blockchain systems is decentralized finance (DeFi). The term describes various financial products and protocols running over a decentralized blockchain system. Examples of such protocols are exchanges, lending and borrowing, and derivatives. Due to their open and decentralized nature, permissionless blockchains have the potential to democratize the financial market, as every party can participate in the financial market, even playing roles that are reserved for banks or big financial institutes in traditional finance.

1.1. Open Challenges

In order to understand the state-of-the-art of blockchain technology, we analyzed the decentralized finance landscape over Ethereum. In our work [4], we identified limited scalability as the major technical challenge hindering mass adaption. With an increasing interest in the system, the number of participants and transactions grows. Eventually, this results in a congested network since it cannot handle that many transactions. A congestion happened to the Ethereum system in 2017 when the CryptoKitties game became highly attractive and significantly slowed down the entire system [3]. Similarly, the Bitcoin system became clogged in 2023 when the interest in BRC-20 tokens increased resulting in an increase of transaction fees by over 300% [3]. These examples show that new techniques need to be developed and deployed to meet the increasing demand.

Another open challenge of blockchains is the limited privacy features. Due to the public nature of many blockchain systems including Bitcoin and Ethereum, the money transfers and smart contract executions are inherently public. This information disclosure poses a severe privacy threat, since transactions including their inputs and results are visible to everyone. While Bitcoin and Ethereum were designed without privacy at first, other systems like Monero [175] and Zcash [25] aim to solve this issue and provide some privacy guarantees. However, Bitcoin and Ethereum are still the most popular systems by now. Therefore, it is desirable to add privacy guarantees on top to these systems.

Finally, more capabilities lead to more applications of blockchain systems. In particular, blockchains are linked together with other components. For instance, the self-sovereign identity (SSI) context combines blockchains with the concept of decentralized identities and anonymous credentials [157]. Often, these components are not explicitly designed to be used with blockchains. As a result, they might not adhere to the fundamental idea of blockchain systems, meaning trusted third parties are replaced by a decentralized system. Therefore, effort needs to be taken to align these components with the decentralized setting of blockchains.

In total, blockchain technology requires new techniques and solutions for the three open problems of limited scalability, restricted privacy, and decentralization of new blockchain applications. We elaborate on all of these challenges in the following.

Scalability

In order to post a transaction on the blockchain, users send their transactions to miners who maintain the system. Miners store the users' transactions first in the mempool. Then, a miner bundles a bunch of transactions into a block. Next, the block is appended to the previous chain. Depending on the consensus algorithm, a miner needs to take different actions. In the case of Bitcoin, whose consensus algorithm uses PoW, a miner needs to solve a cryptographically hard puzzle. Once the puzzle is solved, the miner distributes the new block to the entire network. All other miners can verify the block's validity and the puzzle's solution. If both are valid, the new block is appended to the previous chain, and all miners try to append the following block to the new one. In Bitcoin, the parameters of the cryptographic puzzle are set and automatically adjusted such that a new block is mined roughly every ten minutes [32]. Ethereum is based on a different consensus algorithm than Bitcoin and has an average creation time of 12 seconds [32]. Both have in common that the block size is fixed, and it takes a predefined amount of time before a new block is created. Given a fixed block size and a predefined block creation time, the number of transactions per second is limited. For instance, in Bitcoin, the theoretical block size is limited to 4 MB [1] and the block creation time is ten minutes. Given a transaction size of 192 bytes for a simple payment [190], the maximum number of transactions per second is 36.[2]

If the number of transactions sent to the network exceeds the maximal throughput, the mempool increases, and congestion happens. Congestion might have a severe impact, ranging from missed timeouts to a dramatic increase in transaction

[1]Since 2017, Bitcoin uses a block weight instead of a fixed block size [200]. For simplicity, we consider the theoretical block size, which is accurate enough to compare the magnitudes.

[2]We stress that this calculation is strongly simplified but good enough to show the magnitude. We refer the reader to [104] for more details.

fees to incentivize the miners to include one's transaction before other transactions with lower fees.

Compared to centralized systems such as VISA, which achieves a throughput of around 65,000 transactions per second [192], we must confess that blockchain systems are not competitive without additional techniques. Therefore, new approaches and techniques must be considered to provide better scalability for blockchain systems.

Privacy

Recall that the fundamental idea of decentralized cryptocurrencies is eliminating trusted parties. This idea should also hold for checking the validity of transactions. In particular, every party should be able to check a transaction's validity independently. Many blockchain systems, including Bitcoin and Ethereum, provide this property by making all data publicly visible. The public data includes all payment details like sender, amount, and receiver and all inputs to smart contract executions. In total, the entire transaction history is publicly available. This fact allows every party to check invalid transactions such as double spending of coins or incorrect smart contract execution. However, exposing all data to the public poses a serious privacy threat. In particular, everyone can see to which address money is transferred and which smart contract is executed. To illustrate the negative impact, consider a smart contract that gives money to someone who solves a mathematical problem. Once a party solves the puzzle and sends a transaction to the network, anyone else who sees the transaction and, thus, the solution can create their own transaction containing the solution and try to include this transaction before the benign transaction.

All in all, privacy is an important feature. However, many blockchain systems were designed with little privacy at first. The most popular blockchains, Bitcoin and Ethereum, have only little privacy features built-in, e.g., pseudonymity. However, there are deanonymization attacks to bypass this feature [27, 126]. Therefore, adding an extra layer of privacy protection is especially attractive.

Decentralization in Applications

Since the emerge of Bitcoin, blockchains were first used to facilitate simple payment systems allowing parties to transfer coins. Later, Ethereum introduced the execution of smart contracts. These contracts allow to run arbitrary code on a blockchain facilitating conditional payments based on programmed behavior and much more. The combination of blockchains and smart contracts opened blockchains for a huge range of new applications [194].

As blockchains gain more and more attraction, the number of applications increase steadily. Moreover, research and industry become more interested in this technology. Consequently, blockchains are used in new contexts such as Industry 4.0 [62]. Another particularly interesting context is self-sovereign identities (SSI) [35]. The idea behind SSI is to provide every individual the opportunity to manage their own identity. This opportunity includes the power to decide when and which information are revealed to other parties. SSI is build on top of three components [177]. First, blockchain provides a decentralized infrastructure to store and manage public metadata. Second, the concept of decentralized identifiers defines a framework and standard of how to address parties. Third, anonymous credentials allow parties to show credentials without revealing more information than desired. For instance, parties can prove they are authorized to access special content on some website without revealing anything else.

As a blockchain provides the infrastructure, the system profits from its decentralized nature. However, there is a discrepancy as other building blocks still rely on trusted parties. For instance, to issue anonymous credentials, often a central issuer is required. It is desirable to extend the decentralized setting to components that are paired up with blockchains to remove trusted parties.

1.2. Goal of this Thesis

This thesis aims to provide new solutions for the highlighted challenges of blockchain systems. Secure multiparty computation (MPC) is an essential building block in all our solutions. More concretely, our contribution is threefold. First, we use MPC in combination with trusted execution environments (TEEs) to *solve drawbacks of existing scalability solutions.* While TEEs offer fast and confidential execution, we use MPC to add redundancy and, thus, increase the availability of our solution. Second, we provide a *new approach to privacy-preserving smart contract execution.* While MPC inherently offers private computation, we present more efficient protocols with adequate security guarantees. Third, we *apply the decentralization paradigm of blockchains to a popular anonymous credential scheme.* To this end, we design a tailored MPC protocol to issue credentials distributively and efficiently. Since MPC is the primary building block of all our contributions, we provide a high-level introduction to MPC, highlight its promising features, and stress the challenges of using MPC for our goals.

Secure Multiparty Computation

Secure multiparty computation (MPC) allows distrusting parties to compute a predefined function on private inputs. The term comprises tailored protocols, which compute a specific function, such as the signing algorithm with the secret key being split between a set of keyholders, and general-purpose protocols, which allow the computation of arbitrary functions. Independent of the function, secure MPC protocols provide correctness and privacy. Correctness means that the output of the protocol execution is equal to the function's output on the same input. Privacy guarantees that parties cannot learn more about the other parties' inputs except what they can derive from the output itself. The security properties hold against an adversary that can corrupt parties of the protocol execution. Depending on the fraction of the corrupted parties, we distinguish between the honest majority setting, where the adversary is allowed to corrupt less than half of the parties, and the dishonest majority setting. In the latter setting, the adversary can corrupt all but one party. Besides the number of corrupted parties, we distinguish between the types of corruption. Most common are two types of adversaries. On the one hand, an honest-but-curious adversary, also called a semi-honest or passive adversary, instructs corrupt parties to behave like honest parties, but it is trying to learn additional information from the communication. On the other hand, a malicious adversary, also called an active adversary, may instruct the corrupted parties to deviate arbitrarily from the protocol specification. This way, the adversary tries to learn additional information except the output. We denote a protocol being secure in the presence of a semi-honest adversary as semi-honestly or passively secure, and we call it maliciously or actively secure if security holds even in the presence of a malicious adversary. It is easy to see that malicious security provides more robust security guarantees as security holds even in the presence of a stronger adversary. A drawback of the malicious security setting is that we usually require expensive cryptographic means to achieve this level of security. Thus, a semi-honestly secure protocol provides better efficiency at the cost of a lower security level.

MPC is an appealing candidate for achieving privacy of computation. Additionally, decentralization is easy to obtain as MPC allows parties to compute on their inputs jointly. However, MPC comes with significant overhead in terms of computation, communication, and runtime compared to centralized execution. Therefore, we aim to (1) design efficient protocols tailored towards a specific functionality and (2) take a step forward in general-purpose MPC by finding a compromise between security and efficiency.

1.3. Thesis Outline

In Chapter 2, we give the notation we use throughout this thesis. Additionally, we provide backgrounds on blockchains and MPC. Moreover, we present the necessary preliminaries on trusted execution environments (TEE), and important cryptographic building blocks.

In Chapter 3, we present the contribution contained in our publication [99]. Concretely, we advance the research on scalability solutions by presenting a new off-chain protocol. Our protocol combines TEEs and a tailored MPC protocol to provide correctness, privacy, and availability. During the presentation, we detail how our protocol solves drawbacks of previous solutions.

In Chapter 4, we advance the research on private smart contract execution for existing blockchain systems. We detail our contribution in the publications [94, 95, 96] proposing the use of MPC to realize privacy-preserving computing for blockchains. More concretely, we start by considering a middle ground between semi-honest and malicious security called the covert security setting. In [96], we present efficiency optimizations in this setting for an important class of protocols. Moreover, we improve efficiency without weakening the security. Next, we consider an extension of the covert setting which is called publicly verifiable covert security (PVC). In publication [95], we present a generic compiler to publicly verifiable covert protocols. Lastly, in [94], we combine the publicly verifiable covert setting with blockchains. This allows parties to perform computation via an MPC protocol and the result of the computation determines the distribution of coins in the blockchain, analogously to the execution of a smart contract. While malicious behavior during the MPC execution cannot be prevented in the covert setting, we incorporate means to allow honest parties to point out malicious parties towards a smart contract. Our techniques enable automatic verification of the claim inside a smart contract which results in financial punishment of malicious parties and, thus, strengthens the security.

In Chapter 5, we present the contribution contained in publication [93]. Concretely, we present a threshold protocol for creating BBS+ signatures, which are a popular instantiation of anonymous credentials. This way, we extend the decentralized setting to issuing anonymous credentials, an essential building block in the blockchain-related context of SSI.

Finally, in Chapter 6, we present a conclusion and point out interesting directions of future research.

2. Preliminaries

In this chapter, we state the notation used throughout this thesis. Additionally, we recall necessary backgrounds on blockchains (Section 2.1), multiparty computation (Section 2.3), trusted execution environments (Section 2.4), and the BBS+ signature scheme (Section 2.5).

2.1. Notation

Throughout this thesis, we denote the computational security parameter by κ and the statistical security parameter by λ. We define $[k] = \{1, \ldots, k\}$, i.e., the set from one to k. The set from zero to k is explicitly stated by $\{0, \ldots, k\}$. We further use $\vec{x}$ to denote a vector, i.e., $\vec{x} = (x_1, \ldots, x_\ell)$. The operator $\overset{\$}{\leftarrow}$ signals a uniform sampling, e.g., $x \overset{\$}{\leftarrow} \mathbb{Z}_p$ denotes x being uniformly random sampled from $\mathbb{Z}_p$.

2.2. Blockchain

In this thesis, we consider a blockchain as a decentralized data structure maintained by a set of parties called miners. Miners receive transactions from users, pack these transactions into blocks, and perform a consensus algorithm to append a freshly created block to the previous one. This process forms a chain of blocks. As a consequence of many consensus algorithms, e.g., algorithms based on proof-of-work (PoW) [87, 119, 158] and proof-of-stake (PoS) [127], forks of the chain might happen. In this scenario, two different states of the system exist in parallel. The miners use the consensus algorithm to agree on one state. Agreeing on one state makes the transactions included in the discarded block invalid. The opportunity of invalid transactions poses a risk to users since they might take actions based on transactions that will be discarded afterward. Therefore, the concept of confirmation blocks was introduced. The idea behind this concept is that blocks and transactions are considered final as soon as several confirmation blocks are appended afterward. The more confirmation blocks exist, the lower the probability that a block and its transactions will be discarded. Abstractly speaking, a blockchain can be considered an append-only ledger by considering only finalized transactions.

In this thesis, we focus on blockchains that support smart contract execution, e.g., Ethereum [203]. Therefore, transactions created by users can either be simple payments or more complex smart contract invocations.

2.3. Multiparty Computation[1]

Secure multiparty computation in the standalone model is defined via the real-world/ideal-world paradigm. In the real world, all parties interact in order to execute the protocol π jointly. In the ideal world, the parties send their inputs to a trusted party called ideal functionality, denoted by $\mathcal{F}$, which computes the desired function f and returns the result to the parties. Since $\mathcal{F}$ is trusted, the ideal world guarantees the correctness of the computation 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 $f = (f_1, \ldots, f_n)$, be the function realized by π. For every input vector $\vec{x} = (x_1, \ldots, x_n)$ the output vector is $\vec{y} = (f_1(\vec{x}), \ldots, f_n(\vec{x}))$ and the i-th party P_i with input x_i obtains output $f_i(\vec{x})$.

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

2.3.1. Covert Security

Aumann and Lindell introduced the notion of *covert security with ϵ-deterrence factor* in 2007 [13]. We focus on the strongest given formulation of covert security, which is the *strong explicit cheat formulation*, where the ideal-world adversary only learns the honest parties' inputs if cheating is undetected. However, in this thesis, we consider a slightly modified version of the original notion of covert security to

[1]This section is taken verbatim from our publication [95] with minor adjustments.

capture realistic effects that occur, especially in input-independent protocols, and are disregarded by the notion of [13]. The changes are detailed and explained in our publication [95].

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, called the deterrence factor. Let $\epsilon : \mathbb{N} \to [0,1]$ 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 $\texttt{ok}_i$ instead of their inputs to the trusted party.

Trusted party answers adversary: If the trusted party receives inputs from all parties, denoted by $\vec{x}$, the trusted party computes $(y_1, \ldots, y_n) = f(\vec{x})$ 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 $w_i = \texttt{corrupted}_i$ to the trusted party, the trusted party sends $\texttt{corrupted}_i$ to all honest parties and halts. If multiple parties send $\texttt{corrupted}_i$, then the trusted party disregards all but one of them (say, the one with the smallest index i). If both $\texttt{corrupted}_i$ and `abort` messages are sent, the trusted party ignores the $\texttt{corrupted}_i$ message.

Attempted cheat option: If a corrupted party sends additional input $w_i = \texttt{cheat}_i$ to the trusted party (as above: if there are several messages $w_i = \texttt{cheat}_i$ 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 $\texttt{corrupted}_i$ 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 $\{x_j\}_{j \notin I}$. Following this, the adversary sends the trusted party `abort` or output values $\{\hat{y}_j\}_{j \notin I}$ 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 $\hat{y}_j$ to P_j.

The ideal execution then ends at this point. Otherwise, if no w_i equals $\texttt{abort}_i$, $\texttt{corrupted}_i$ or $\texttt{cheat}_i$, the ideal execution continues below.

Trusted party answers honest parties: If the trusted party did not receive $\texttt{corrupted}_i$, $\texttt{cheat}_i$ 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 output 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 $\mathsf{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),I}(\vec{x}, 1^{\kappa})$ the output of the honest parties and the adversary in the execution of the ideal model as defined above, where $\vec{x}$ 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 $\vec{x} \in (\{0,1\}^*)^n$, and every auxiliary input $z \in \{0,1\}^*$:*

$$\{\mathsf{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),I}(\vec{x}, 1^{\kappa})\}_{\kappa \in \mathbb{N}} \overset{c}{\approx} \{\mathsf{REAL}_{\pi,\mathcal{A}(z),I}(\vec{x}, 1^{\kappa})\}_{\kappa \in \mathbb{N}}$$

2.3.2. Covert Security with Public Verifiability

Asharov and Orlandi introduced the notion of *covert security with ϵ-deterrent and public verifiability* (PVC) in the two-party setting [10]. We give an extension of their formal definition to the multiparty 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 $\texttt{corrupted}_j$ in the protocol execution for $j \in I$ and returns a certificate `cert`, i.e., $\texttt{cert} := \mathsf{Blame}(\texttt{view}_i)$. The Judge-algorithm takes as input a certificate `cert` and outputs the identity $\texttt{id}_j$ 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 so that an honest party P_i computes $\texttt{cert} = \mathsf{Blame}(\texttt{view}_i)$ and broadcasts it after detecting cheating. We denote the modified protocol by π'. Notice that the adversary gets access to the certificate due to this change. By requiring simulatability, the certificate is guaranteed not to reveal any private information.

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

Definition 2 (Covert security with ϵ-deterrent and public verifiability in the multiparty case (PVC-MPC))**.** *Let* f, π', Blame*, and* Judge *be as above. The triple* $(\pi', \mathsf{Blame}, \mathsf{Judge})$ 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 probabilistic polynomial time adversary $\mathcal{A}$ corrupting parties P_i for $i \in I \subseteq [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\vec{x}, z) \in (\{0,1\}^*)^{n+1}$ the following holds:*
   If $\mathsf{OUTPUT}_j(\mathsf{REAL}_{\pi,\mathcal{A}(z),I}(\vec{x}, 1^\kappa)) = \mathtt{corrupted}_i$ *for $j \in [n] \setminus I$ and $i \in I$ then:*
   $$\Pr[\mathsf{Judge}(\mathtt{cert}) = \mathtt{id}_i] > 1 - \mu(n),$$
   where $\mathtt{cert}$ *is the output certificate of the honest party P_j in the execution.*

3. *(Defamation Freeness) For every probabilistic polynomial time adversary $\mathcal{A}$ corrupting parties P_i for $i \in I \subseteq [n]$, there exists a negligible function $\mu(\cdot)$ such that for all $(\vec{x}, z) \in (\{0,1\}^*)^{n+1}$ and all $j \in [n] \setminus I$:*
   $$\Pr[\mathtt{cert}^* \leftarrow \mathcal{A}; \mathsf{Judge}(\mathtt{cert}^*) = \mathtt{id}_j] < \mu(n).$$

2.4. Trusted Execution Environments[2]

We comprise the hardware and software components required to create confidential and integrity-protected execution environments under the term trusted execution environment (TEE). An operator can instruct its TEE to create new *enclaves*, i.e., new execution environments running a specified program. We follow the approach of Pass et al. [163] to model the TEE functionality. We briefly describe the operations provided by the ideal functionality formally specified in [163, Fig. 1]. A TEE provides a $\mathsf{TEE.install}(\mathtt{prog})$ operation which creates a new enclave running the program $\mathtt{prog}$. The operation returns an enclave id $\mathtt{eid}$. An enclave with id

[2]This section is partly taken verbatim from our publication [99] with minor adjustments.

`eid` can be executed multiple times using the $\mathsf{TEE.resume}(\texttt{eid}, \texttt{inp})$ 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. [163] and include an attestation mechanism provided by a TEE that generates an attestation quote ρ over $(\texttt{eid}, \texttt{prog})$. ρ can be verified by using method $\mathsf{VerifyQuote}(\rho)$.

Our results within this thesis are based on specific correctness and privacy guarantees of TEEs. In particular, we require TEEs to provide integrity and confidentiality guarantees for all its enclaves meaning that the enclave program runs correctly and does not leak any data. Privacy holds specifically for all cryptographic material stored inside an enclave. Additionally, we assume TEEs provide a good source of randomness and a relative clock to all its enclaves.

2.5. BBS+ Signature Scheme[3]

Bilinear maps. We briefly recall the basics of bilinear maps following [38, 40]. Let BGen be a randomized algorithm that on input a security parameter κ outputs a prime p, such that $\log_2(p) = O(\kappa)$, three cyclic groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order p, generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$, and a pairing $\mathsf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.

We call e a bilinear map if the following properties hold:

- Bilinearity: For all $u \in \mathbb{G}_1$, $v \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$, we have $\mathsf{e}(u^a, v^b) = \mathsf{e}(u, v)^{ab}$.
- Non-degeneracy: For generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ it holds that $\mathsf{e}(g_1, g_2) \neq 1$. Since $\mathbb{G}_T$ is of prime order p, this implies that $\mathsf{e}(g_1, g_2)$ is a generator of $\mathbb{G}_T$.
- Efficiency: e can be computed efficiently.

The literature differentiates between three types of pairings [101]: Type-1 with $\mathbb{G}_1 = \mathbb{G}_2$, Type-2 with $\mathbb{G}_1 \neq \mathbb{G}_2$ and the existence of an efficiently computable isomorphism $\phi : \mathbb{G}_2 \to \mathbb{G}_1$, and Type-3 with $\mathbb{G}_1 \neq \mathbb{G}_2$ and no such isomoprhism ϕ.

BBS+ signature scheme. The BBS+ signature scheme allows to create signatures on an array of message with constant size. Let k be the size of the message arrays, $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, \mathsf{e})$ be a bilinear mapping tuple obtained form BGen defined above and $\{h_\ell\}_{\ell \in \{0,\ldots,k\}}$ be random elements of $\mathbb{G}_1$. The BBS+ signature scheme is defined as follows:

[3]This section is taken almost verbatim from our publication [93].

- $\mathsf{KeyGen}(\kappa)$: Sample $x \overset{\$}{\leftarrow} \mathbb{Z}_p^*$, compute $y = g_2^x$, and output $(\mathsf{pk}, \mathsf{sk}) = (y, x)$.
- $\mathsf{Sign}_{\mathsf{sk}}(\{m_\ell\}_{\ell \in [k]} \in \mathbb{Z}_p^k)$: Sample $e, s \overset{\$}{\leftarrow} \mathbb{Z}_p$, compute $A := (g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$ and output $\sigma = (A, e, s)$.
- $\mathsf{Verify}_{\mathsf{pk}}(\{m_\ell\}_{\ell \in [k]} \in \mathbb{Z}_p^k, \sigma)$: Output 1 iff $\mathsf{e}(A, y \cdot g_2^e) = \mathsf{e}(g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell}, g_2)$

The BBS+ signature scheme is proven strongly unforgeable under the q-strong Diffie-Hellman assumption (q-SDH) for pairings of type 1, 2, and 3 [12, 53, 187]. Intuitively, strong unforgeability states that the attacker is not possible to come up with a forgery even for messages that have been signed before. We refer to [187] for further details.

Recently, Tessaro and Zhu showed an optimized version of the BBS+ signatures, reducing the signature size [187]. In their scheme, the signer samples only one random value, $e \overset{\$}{\leftarrow} \mathbb{Z}_p$, computes $A := (g_1 \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$, and outputs $\sigma = (A, e)$. The verification works as before, with the only difference that the term h_0^s is removed. Note that if the first message m_1 is sampled randomly, then the short version is equal to the original version.

3. New Off-Chain Protocol for Smart Contract Execution

In the Introduction, we highlighted the necessity of scalability solutions for popular blockchain systems. In this section, we state existing approaches, mention their drawbacks, and present a new scalability solution using a tailored multiparty computation (MPC) protocol.

We categorize existing solutions based on different characteristics. First, we distinguish between *on-chain* and *off-chain* solutions. On-chain solutions improve throughput and transaction speed by changing the blockchain system itself. On the one hand, such changes include modifications of blockchain parameters like the block creation time or the block size. On the other hand, changes to the consensus algorithm of the blockchain system can improve the scalability. The most popular on-chain scalability approach is *sharding*, where the state of the blockchain is split [193]. For every split of the state, called shard, only a subset of miners perform the validation. This fragmentation enables parallel processing of transactions by modifying different shards.

The main characteristic of on-chain solutions is the modification of the blockchain system. However, changes to the underlying technology require either a completely new system or all parties of the system must adapt to the modification. Since it is likely that not all parties agree to the changes, a fork of the system might occur, splitting the user base into two parts. This scenario happened when a set of developers proposed to increase the block size of Bitcoin from 1MB to 8MB [198]. Since not all parties agreed to the suggested change, a fork occurred on August 1, 2017, and Bitcoin Cash was created [31]. As the consequences of such modifications might be drastic, the on-chain approach is less popular than off-chain solutions.

In contrast to on-chain solutions, *off-chain* protocols extend existing blockchain systems without modifying the underlying technology. Instead, these protocols allow users to offload transactions from the blockchain and thus reduce the computational demand of the blockchain. In order to use off-chain protocols, a fork of the system is avoided, but parties can make on-chain and off-chain transactions simultaneously.

Abstractly speaking, to use off-chain protocols, users first make an on-chain transaction to transfer money to the off-chain system. Next, they use the off-chain protocol, and finally, they make an on-chain transaction again to transfer money back from the off-chain system to the blockchain. While all off-chain solutions follow this blueprint, the concrete techniques and details differ widely.

Before giving examples of off-chain protocols, we make another categorization. We categorize off-chain protocols based on the supported operations. On the one hand, some systems allow users to transfer money only. On the other hand, systems exist that allow users to perform more complex transactions, including the execution of smart contracts. In this thesis, we focus on off-chain protocols for smart contract execution.

In the last decade, several off-chain protocols for smart contract execution have been proposed. There exist different approaches including state channels [90], plasma chains [125, 168], rollups [8, 160, 161, 189], and trusted execution environment (TEE)-based protocols [63, 79].

A significant challenge of off-chain protocols is guaranteeing that users can interact with the system and execute contracts anytime. This desired property is called availability or liveness. The technical challenge to achieve this property is that protocols must handle unresponsive parties. Here, we focus on the situation where parties intentionally stop responding, i.e., behave maliciously, and disregard unintentional stoppage, e.g., due to power outages. Existing scalability solutions build on different techniques to achieve liveness, where different approaches incur different drawbacks.

One way to deal with unresponsive parties is to require all parties to lock collateral. In case of an abort, other parties get compensated using the locked money. Consider the scenario where one party expects to lose money if the smart contract execution terminates, e.g., the party expects to lose a poker game. Even if that party decides to stop participating in the off-chain protocol to prevent the contract execution from proceeding, other parties are compensated, and the malicious party loses its collateral. The drawback of this approach is that the parties need to lock a huge amount of money. The exact amount depends on the group of participants, so the participants must be fixed for the contract's execution. Furthermore, the contract's lifetime must be fixed at the onset to guarantee the compensation's payout.

Another way to deal with unresponsive parties is to store the state of the off-chain smart contract on the blockchain. This enables other parties to carry on the contract execution in case the intended party is unresponsive. Obviously, this approach suffers from regular on-chain transactions, which are expensive. Another drawback is that the contract state cannot be kept confidential if stored on the blockchain so that other parties can continue the execution. This lack of a private state restricts the class of supported contracts.

We summarize the following limitations and show in Section 3.2 that all existing off-chain protocols for smart contract execution suffer from at least one of them:

1. **High collateral:** Participating parties must lock a high amount of money to compensate other parties in case of an abort.

2. **Fixed set of participants and lifetime:** The set of parties and the contract lifetime must be fixed at the beginning of the contract execution.

3. **Regular on-chain transactions:** The protocol requires regular costly on-chain transactions.

4. **No support of private state**: The contract state is not confidential, and thus, contracts that require a private state are not supported.

3.1. Our Contribution

In this thesis, we contribute to the research field of off-chain scalability solutions for smart contract execution. To this end, we present a new off-chain protocol called *POSE - Practical Off-chain Smart contract Execution* in the following publication, which can be found in Appendix A:

[99] 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**.

Concretely, in the above publication, we present an off-chain protocol for smart contract execution that solves all of the aforementioned drawbacks 1-4 at the same time. Our protocol is based on TEEs and a tailored multiparty protocol. POSE enables a single operator to perform the contract execution and to sync the state updates with a pool of watchdogs. In case the executing operator is not responsive, one of the watchdogs takes over the execution. This way, our protocol achieves liveness even if a large fraction of parties is malicious. Consequently, our protocol avoids high collateral and regular on-chain transactions in the optimistic case and supports unlimited lifetimes and participants. Additionally, the use of TEE provides state confidentiality during the execution, and our protocol guarantees that parties can only learn information about the state if all parties eventually learn the information.

In the following, we first present the protocol architecture and state the considered adversary model, then we give a high-level view of the POSE protocol, and finally, we take a closer look at the mechanisms to achieve liveness and private state.

3.1.1. Protocol Architecture and the Adversary Model

Our POSE system consists of three types of parties: users, operators, and a single manager contract running on the blockchain. This smart contract combines the function of a bridge, i.e., it provides the entry point to the off-chain system for users, a registry of operators, and a referee in case of challenges due to unresponsive parties. Users aim to use the system to execute smart contracts cost-efficiently. Operators possess TEEs and contribute computing power to the system.

We consider a static malicious adversary who may corrupt an arbitrary number of users and a fixed fraction of the operators. Corrupted parties may deviate arbitrarily from the protocol specification and, in particular, may abort at any time. As stated in Section 2.4, we consider a TEE to provide a secured environment to perform computation. The security properties imply that corrupted operators cannot learn information from the computation executed inside the TEE, and they cannot read information from their TEE's storage. In particular, any cryptographic material, such as encryption keys, is hidden from the adversary. However, the operator is under full control of its TEE's communication channels. Thus, a corrupted operator can intercept, delete, insert, and replay any messages from and to its TEE. Our POSE protocol incorporates standard cryptographic means such as signatures and encryption to facilitate authenticated and secure communication over these channels.

3.1.2. Protocol Overview

Our POSE protocol consists of several subprotocols, which we present below. First, operators need to register in the POSE system. Then, users can create new smart contracts within POSE. Finally, users executed existing contracts. Figure 3.1 illustrates the contract creation with step 1-4 and the optimistic contract execution with step 5 and 6. While the contract creation and execution require only one on-chain transaction in the optimistic case, our protocol must cope with situations where operators are unresponsive. To this end, another subprotocol is a challenge and response procedure, where a dependent party challenges an unresponsive party on-chain. This subprotocol requires two on-chain transactions—one challenge message and one response message.

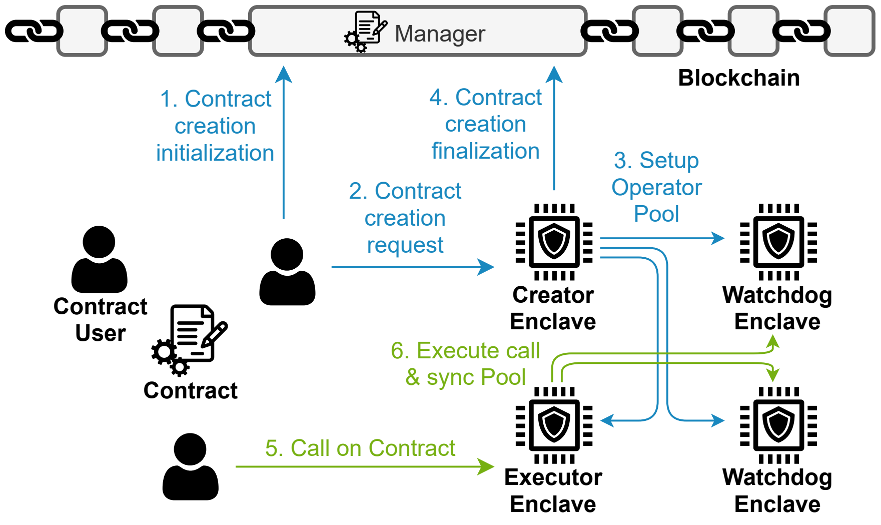


Figure 3.1.: The contract creation (steps 1-3) and execution (steps 5 and 6) within the POSE system. The figure is taken from our publication [99].

Operator registration. An operator willing to contribute its TEE's computing power to the POSE system starts the *registration* process. This process involves the operator, its TEE, and the on-chain manager contract. The goal of this process is to register the identity of a freshly generated TEE enclave in the manager, attest that the TEE enclave is running a specific POSE program, and ensure that the enclave is aware of the blockchain data up to some fixed offset.

Initially, the operator instructs its TEE to create a new enclave running a specific POSE program. The program comprises all the contract-independent code necessary to maintain the system, e.g., to receive state updates if the enclave works as a watchdog. When creating a new enclave, the enclave generates a public and private key pair (pk, sk), where the public key is returned to the operator, and the secret key is kept confidential in the enclave storage. The public key is later used to identify this enclave in the list of all registered enclaves. Next, the operator uses the attestation mechanism of its TEE to attest that the freshly generated enclave is running the POSE program and that pk belongs to sk. Finally, the operator provides blockchain data to the enclave, which returns a signature on the information received. This step is necessary to ensure the enclave knows the blockchain data up to some fixed offset. We elaborate more on the blockchain data when explaining the synchronization mechanism in Section 3.1.4.

After the operator obtains the public key, the attestation, and the signature on the blockchain data from the enclave, the operator provides the data to the manager contract. The manager checks that the attestation and the signature are valid and that the signed blockchain data is up to date. If the checks hold, the manager adds the public key of the enclave to the list of registered enclaves. Eventually, the newly generated enclave is registered and ready to participate in off-chain smart contract execution.

Smart contract creation. The creation of a smart contract within the POSE system is initiated by a user who wants to deploy the contract. To this end, the user selects an arbitrary enclave, called the creator enclave, from the list of all registered enclaves. Then, the user sends the creator enclave's identity together with the contract code's hash to the on-chain manager. The manager allocates storage for the contract information and marks the contract as "in creation". Next, the user sends a creation request, including the contract code, to the creator enclave. Upon receiving a creation request, the creator enclave randomly selects a set of registered enclaves. This set forms the contract pool from now on. Such a pool consists of a dedicated executor enclave and several watchdogs. While the executor enclave is responsible for performing the contract execution, the watchdogs receive state updates from the executor and take over the execution in case the executor is unresponsive. In addition to selecting the pool, the creator enclave generates a symmetric encryption key to facilitate efficient secure communication between the pool members. The creator enclave distributes[1] the information, i.e., the contract code, the pool, and the key, to all pool members, which locally set up the smart contract. After receiving confirmations from all pool members, the creator enclave creates a signed creation confirmation. This signed message is sent to the manager, who marks the contract as "initialized" after successful validation.

In the optimistic case, where every enclave and its operator respond in time, the creation protocol requires two on-chain transactions. We can even improve this to only one on-chain transaction if the user omits the initial transaction to the manager. Since this message is only required to start a challenge afterward, it is unnecessary in the optimistic case. In case some parties are unresponsive, either the creator enclave or some of the pool members, the dependent party, i.e., the user or the creator enclave, starts a challenge-response protocol. In the worst case, the parties execute at most two challenge-response protocols, resulting in four sequential on-chain transactions. We explain the challenge and response procedure below.

[1]The creator enclave uses the public keys of the pool members to distribute the information confidentially.

Contract execution. The contract execution protocol is initiated by a user who wants to execute an existing smart contract. The user reads the identity of the executor enclave from the on-chain manager contract. Then, it sends an execution request to the executor containing the input to the smart contract and a nonce. The nonce prevents replay attacks, where a corrupted party tries to execute the move twice. Upon receiving an execution request, the executor enclave executes the user's move. Next, the executor enclave distributes the updated state to all the watchdogs, which update their internal state and respond with a confirmation message. After receiving a confirmation from every watchdog, the executor sends the public state back to the user. Note that state updates transferred to the watchdogs are encrypted under the symmetric pool key, which was set up during the contract creation protocol. Therefore, even a malicious operator who controls the connection between enclaves can only learn information once the executor outputs the new public state.

Again, we described the optimistic case, where all enclaves were responsive. In the case of unresponsive parties, either an unresponsive executor or watchdogs, the dependent party starts a challenge period. If the executor does not respond to a challenge, it will be kicked out of the pool, and one of the watchdogs will become the new executor. The user can then send a new execution request to the new executor. If one or multiple watchdogs are unresponsive, the executor launches a challenge. Then, all watchdogs who do not respond to this challenge are kicked out of the pool. As a result, all watchdogs still in the pool possess the state updated and send a confirmation. We elaborate on the challenge-response mechanism below.

Challenge and response procedure. In the optimistic case, where all parties act in time, most of the messages are sent off-chain, including all messages of the execution protocol. Only the creation protocol requires on-chain transactions to inform the on-chain manager about a freshly instantiated smart contract. Unfortunately, the protocol must care about unresponsive parties, where the operators decide to stop acting on requests. In order to still guarantee an ongoing protocol, we incorporate a challenge-response protocol. In this protocol, the dependent party acts as the challenger, and the challenged party must respond. While there are several occasions where a challenge-response protocol can be executed, we exemplify the procedure when the executor is not responding. In this scenario, the user is the challenger.

The user starts the challenge period by sending an on-chain transaction to the manager. This message contains the intended move and is stored by the manager. Additionally, the manager starts a timeout when the executor must respond. The operator of the executor enclave monitors the on-chain manager to identify challenges. Upon seeing a challenge, the operator sends the move inside the challenge to its enclave to execute the smart contract. The execution works exactly as highlighted above, i.e., the executor enclave updates the contract state and propagates the state update to all watchdogs. Eventually, the executor enclave returns a response containing the updated public state. This response is forwarded to the on-chain manager by the enclave's operator. Since the response contains the updated public state, the user can read the result of the smart contract execution from the response. Hence, this sequence yields a successful contract execution.

If the response is not submitted in time, the user triggers the manager to kick the current executor enclave from the pool. Following that, one of the watchdogs takes over the executor role. Note that the timeout parameter must be defined to give the executor enough time to answer any challenge. This property is also necessary to thwart a dishonest user kicking a responsive executor. When defining the timeout parameter, we must consider that the executor might need to start a challenge period by itself. Concretely, the executor creates a new on-chain challenge if some watchdogs do not reply with a state update confirmation. Therefore, the timeout parameter for an executor challenge must be high enough so no honest executor can be kicked due to malicious behavior of the user or the watchdogs.

3.1.3. High Availability Guarantees

The core of our POSE system is an MPC protocol to sync state updates between a set of enclaves. This pool is the primary mechanism for achieving high availability guarantees. If the executor enclave stops responding, one of the watchdogs takes over the execution. In the following, we explain the availability guarantees provided by POSE in more detail.

We start examining the contract creation procedure. Here, the creator enclave selects a random subset of all registered enclaves of a configurable size. Since we consider a static adversary who controls a fixed fraction of operators and the selected operators are random, the adversary cannot bias the pool selection. Additionally, a single honest operator is enough to ensure the liveness of the contract execution. This low requirement is because only a single responsive operator is required to act as an executor. Even if only one operator continues responding and all other operators stop reacting, this honest operator takes over the executor's role. Additionally, the POSE protocol is designed so that the adversary cannot kick an honest operator out of the pool. In particular, the timeouts of the challenge protocols are set such that an honest executor has enough time to respond. This requirement holds even if the watchdogs are malicious and abort.

We illustrate the scenario for a specific example. Let's consider a pool size $s = 10$, a total of $n = 1000$ operators, and the adversary controls 25% of them, i.e., $m = 250$. Liveness of a single contract holds as long as at least one honest operator is within the contract pool. In contrast, liveness breaks if only malicious operators are selected. The probability for a liveness break can be calculated by $\Pi_{i=0}^{s-1} \frac{m-i}{n-1}$. It follows that the probability of liveness is the inverse probability given by $\epsilon = 1 - \Pi_{i=0}^{s-1} \frac{m-i}{n-1}$. We get the lower bound $\epsilon = 1 - \Pi_{i=0}^{s-1} \frac{m-i}{n-1} > 1 - (\frac{m}{n})^s$. For the given scenario, we obtain a liveness probability for one contract of $\epsilon > 1 - (\frac{250}{1000})^{10} > 99\%$.

To summarize, POSE takes a new approach to guaranteeing high availability. POSE uses a pool of operators together with a tailored MPC protocol. While this requires interaction between the parties, we emphasize that this interaction occurs almost entirely off-chain. The advantage of this approach is that POSE does not require high collateral or regular on-chain transactions. Additionally, POSE supports a dynamic group of participants and an unlimited lifetime.

3.1.4. Supporting Private State

The POSE system supports private states through an interplay of different techniques. While the TEE properties guarantee that an operator cannot learn state information during computation and at rest, we use symmetric encryption to keep state updates confidential in transit. The primary technical challenge to support private states is to reveal information about state updates only if these updates cannot be reverted. Without this property, a malicious operator could trigger a state update with some input, observe the result, revert the state, and decide on its input again. This hypothetical sequence would give the adversary an undesired advantage.

The rollback attack is prevented in the POSE system by two aspects. First, state updates are only revealed to the operator by the executor enclave if all available watchdogs confirm the state update. Consequently, even if the executor enclave is kicked afterward, the watchdog that takes over is aware of the updated state. Here, we must prevent the attack where an operator pretends to its enclave that some watchdogs are being kicked when, in fact, they are not. The only scenario in which an operator gets kicked from a contract pool is when the operator is not responding to an on-chain challenge. Therefore, we need a mechanism to synchronize the state of the on-chain manager with the enclaves. This synchronization is the second aspect required to prevent a rollback attack effectively. As the synchronization mechanism is a crucial aspect of the POSE system, we dive into more details below.

Synchronization

While the contract execution proceeds entirely off-chain in the optimistic case, there are on-chain events of which the executor enclave must be aware. Such events are challenges to which the executor must react, as well as on- and off-ramping transactions. An on-ramping transaction is an on-chain transaction where the user transfers coins to a POSE off-chain contract. The transfer is done by locking coins linked to a specific POSE contract in the manager contract. The executor enclave of this POSE contract monitors the relevant on-chain transactions and transfers the locked on-chain coins into coins usable in the off-chain contract. In order to withdraw coins from an off-chain contract, the user requests an off-ramping transaction from the executor enclave. Such an off-ramping transaction can be posted to the on-chain manager contract to unlock coins. Once an off-ramping transaction is requested, the executor enclave removes the coins from the off-chain balances. This way, users can translate on-chain coins into off-chain coins and vice versa, while the POSE system prevents double-spending.

In order to provide all relevant on-chain data to the executor enclave, we incorporate a specific synchronization mechanism. The mechanism's goal is, first, to provide all relevant on-chain events to the executor enclave and, second, to facilitate a lightweight validation of blockchain data inside the enclave.

In order to achieve the first goal, the executor enclave must receive all relevant blockchain data from its operator. This requirement implies that the provided data is up to date, i.e., the most recent block provided must not be older than a limited offset, and the data is consistent with the blockchain instead of being from a fake chain. We tackle these requirements with several steps. First, during the registration process, the operator provides the latest blocks to its enclave and gets back a signature on the provided data. Then, the signature is forwarded to the manager contract. The manager validates the signature and checks that the signed blocks are part of the blockchain and, at most, a fixed offset behind the most current block. Note that the manager cannot check if the latest signed block is the last on-chain one because the transaction might take some time to be processed. Nevertheless, the offset ensures that the enclave is aware of all on-chain transactions except those that appeared in a limited number of more recent blocks. Next, on every contract execution, the operator provides all new blocks since the last execution. The enclave processes the new blocks to identify relevant on-chain transactions since the last invocation. Additionally, the enclave expects a minimum number of blocks per time interval. This rate is defined by a system parameter and is required so that the enclave stays up to date except for the allowed offset. Eventually, we require that the operator regularly invokes its enclave and provides new blockchain data, even if no contract execution is requested. This requirement is because a malicious operator can create a fake chain with adjusted timestamps if it has enough time to do so. We rule out this attack by requiring regular invocations and a predefined rate of blocks.

Our second goal is to facilitate a lightweight validation of blockchain data inside the enclave. Before, we simplified the description and mentioned that the operator provides blockchain data or blocks to the enclave. However, validating entire blocks inside the enclave is inefficient, and thus, we aim for a better solution.

Instead of entire blocks, the operator provides only block headers and the relevant on-chain transaction with additional information to its enclave. The additional information includes proofs showing that the on-chain transactions are part of the blocks corresponding to the provided block headers. Based on this information, the enclave validates the given block headers as well as the on-chain transactions and incorporates the effects of the on-chain transactions. To prevent the operator from withholding relevant on-chain transactions, we incorporate a hash of all relevant transactions in the on-chain manager state. Consequently, the header verification is only successful if the operator provides all transactions required to recompute the hash value inside the manager state. In total, this mechanism results in a lightweight validation that can be executed inside an enclave.

3.2. Related Work

In the previous section, we presented our POSE protocol to improve the scalability of smart contract-based blockchains. The most popular Blockchain supporting smart contract execution is Ethereum [203], but many other systems, e.g., the BNB Chain [34], Solana [180], Cardano [57] and many more, exist that can benefit from our solution. In this section, we compare our results with other off-chain scaling solutions. We focus on the scaling aspect of POSE and refer to Chapter 4 for related work on privacy-enhancing off-chain solutions. We present related work, highlight the differences to our POSE system, and specifically link back to the drawbacks mentioned in the motivation of this chapter (cf. Section 3).

State channels enable parties to establish a peer-to-peer system to perform transactions off-chain (e.g., [65, 68, 89, 151, 152, 154] and references within [109, 120]). In contrast to payment channels, which support only money transfers, state channels allow the execution of arbitrary computation. A channel consists of an establishment, a transition, and a closure stage. While creating and terminating a channel requires on-chain transactions, the execution of state updates happens completely off-chain. This property yields only constant on-chain transactions in the optimistic case. However, state channels require a fixed set of participants since all involved parties jointly establish a channel. Additionally, the channel's state is known by all parties of the channel to allow them to check the correctness of state updates. This property prevents computation based on private state, like the execution of poker.

Rollups provide a means to improve the throughput of blockchains by performing computation off-chain and reducing storage requirements. A single rollup node bundles transactions and submits the batch to an on-chain contract containing the rollup state. While optimistic rollups consider the published transactions to be valid unless a fraud-proof is submitted [161], zero-knowledge rollups require the rollup node to create a zero-knowledge proof guaranteeing a correct state transition [211]. Rollups are gaining more and more attraction and are implemented by multiple projects, e.g., optimistic rollups by Optimism [160] and Arbitrum One [8] and zk Rollups by StarkNet [183] and zkSync [214]. However, both variants suffer from regular on-chain transactions. Additionally, optimistic rollups publish transactions on-chain in clear and, thus, disclose the state to the public. In contrast, zero-knowledge rollups use validity proofs instead of sending transactions in the clear. However, current implementations of zero-knowledge rollups do not support arbitrary computation due to the lack of efficient zero-knowledge proofs. Note that the rollup operator is aware of the entire state in both variants.

Plasma [168] is similar to rollups because an operator must send regularly state commitments to the blockchain. In contrast to rollups, Plasma suffers from the data unavailability problem, which means that not all data required to verify state updates are available to the users. Moreover, similar to zero-knowledge rollups, Plasma does not support arbitrary computation [166].

Arbitrum [122] and Truebit [188] delegate the computation to off-chain parties, called managers in Arbitrum, while using the blockchain to referee in case of a dispute. In Arbitrum, a set of off-chain managers performs the computation. If they disagree on the outcome of a state update, they perform a bisection protocol to narrow down the disagreement. Then, eventually, an on-chain smart contract validates a small step to resolve the dispute. In contrast to POSE, Arbitrum requires each manager to perform the computation. Additionally, the managers know the entire state, and thus, privacy breaks as soon as one manager is corrupted. Truebit [188] allows parties to pose computation tasks. Then, other parties can solve the task by performing the computation off-chain. Subsequently, the result is posted on-chain. Afterward, a challenge period starts to detect incorrect results submitted by malicious parties. During this period, other parties can challenge the provided result. Similar to Arbitrum, Truebit uses a bisection protocol to resolve the dispute and punish the corrupted party. Truebit suffers from regular on-chain transactions as well as the lack of support for private state.

Bitcontracts [204], ACE [205], and Yoda [80] allow parties to move computation off-chain by assigning a set of providers. Then, state updates are achieved by a quorum of providers agreeing on the same result. Bitcontracts, ACE, and Yoda require the providers to post state updates on the blockchain, resulting in regular on-chain transactions. Additionally, they do not focus on supporting private state. Moreover, Arbitrum, Truebit, Bitcontracts, and ACE have in common that they require trust in the computing operators to achieve correctness of state updates. This is in contrast to POSE, where correctness still holds if all operators are malicious at the cost of requiring trusted execution environments.

FastKitten [79] and Ekiden [63] also leverage TEEs to perform off-chain computation. In contrast to POSE, FastKitten takes only one TEE-equipped operator to perform the computation. Consequently, responsiveness of the operator is financially incentivized via high collaterals. Additionally, the set of parties and the contract's lifetime is fixed to guarantee a payout if the operator is unresponsive. Ekiden achieves liveness by requiring every state update to be submitted on-chain. This procedure allows other parties to pick up the computation at any time. The main drawback of Ekiden is the regular on-chain transaction.

4. Private Computation for Blockchains via MPC

In the Introduction, we emphasize the necessity of enabling private smart contract execution. In this thesis, we focus on adding this feature to existing blockchain systems. For instance, we aim to add privacy-preserving smart contract execution to Ethereum, the most prominent blockchain system that supports smart contract execution but still exposes all data to the public. Compared to designing a new blockchain system from scratch, the advantage of this goal is similar to the benefits of off-chain scalability solutions discussed in Chapter 3. In particular, extending an existing system makes the privacy feature accessible to an existing user base. Extending an existing system prevents the necessity of users switching to a new system, which is a massive hurdle for many users. This burden is evident when user statistics of established systems are compared to follow-up systems. For instance, the number of active addresses in Bitcoin is around 813K as of mid-April 2024 [30]. In comparison, Zcash, a modification of Bitcoin with additional privacy features, has only around 13K active addresses on the same date [209].

Solutions to add private computing to existing blockchains follow different approaches. First, the trusted execution environment (TEE)-based approach builds on a specific hardware component. A TEE is a separate part of computer chips and provides secure enclaves to facilitate confidential computing and storage. The most common TEE is Intel's SGX (software guard extension). Several implementations of this approach were presented in the past, including ShadowEth [208], FastKitten [79], Ekiden [212], Cloak [171], and references within [169]. Note that our POSE off-chain protocol presented in Chapter 3 also supports private smart contract execution and belongs to this class. While this approach is fast, it relies on additional hardware guarantees. The past has shown that attacks on the hardware level exist (e.g., Meltdown [142], Foreshadow [51], Spectre [128]). Additionally, TEEs provide an attestation mechanism to create evidence about the proper operation of a secure enclave. In the case of Intel's SGX, this mechanism requires trust in the manufacturer, constituting a single point of failure and a centralization of risk. For these drawbacks, different cryptographic approaches that rely on computational hardness assumptions instead of hardware guarantees have been proposed.

The second approach builds on fully homomorphic encryption (FHE). This cryptographic primitive allows parties to compute arbitrary circuits on encrypted data. After the evaluation, the final ciphertext can be decrypted to obtain the result. While smartFHE [181] and Pesca [72] provide solutions based on FHE, the former requires still impractical multi-key FHE to support smart contract execution with multiple parties, and the latter builds on an additional trust assumption by requiring trust in the consensus nodes [169].

The third approach builds on zero-knowledge proofs (ZKP). This cryptographic primitive allows a prover to create a proof for some statement. Then, any party can verify the proof without learning anything besides whether or not the statement is true. Several solutions building on ZKPs were introduced in the past (e.g., zkay [185], ZeeStar [184], ZEXE [41], Veri-ZEXE [206], Zapper [186], Kachina [124]). However, these solutions build on trusted setups or are relatively inefficient.

A fourth approach uses multiparty computation (MPC) (e.g., Enigma [215], zkHawk [16], V-zkHawk [17], Eagle [19]). MPC is an appealing candidate for private computation as MPC protocols guarantee that parties learn nothing except what can be derived from the output. However, this approach suffers from several drawbacks. On the one hand, MPC protocols are slow due to their communication complexity compared to the other approaches. On the other hand, solutions based on MPC must deal with malicious parties that deviate from the protocol description. Parties can prove correct behavior via ZKP, but as stated before, this incurs additional drawbacks like the requirement of a trusted setup.

In this thesis, we present a new solution for privacy-preserving smart contract execution via MPC and tackle the challenges above in a new fashion. From a high-level perspective, we cope with the efficiency challenge by considering a slightly weaker security model that facilitates protocols with better efficiency and still an adequate level of security. To deal with malicious parties, we incorporate techniques that enable honest parties to detect malicious behavior and to create a proof of misbehavior. The techniques build on simple cryptographic primitives such as signatures and hash functions instead of requiring heavy cryptographic tools like ZKP.

Before delving into the concrete contributions of this thesis, we illustrate the desired scenario to which our contributions lead. We envision a set of parties that aims to use a smart contract as a conditional payment mechanism, i.e., the contract decides on the distribution of coins based on the parties' inputs and its code. At the onset of the protocol, each party deposits coins in an on-chain manager, which is realized by a separate smart contract. This manager contract is only responsible for distributing the coins and verifying misbehavior, not for performing the computation. Next, the parties execute the contract's code inside an MPC protocol, guaranteeing that the inputs are hidden from other parties. Finally, the parties send the outcome of the protocol to the on-chain contract, which distributes coins accordingly. For instance, the outcome of the protocol can be the winner of some voting poll, and the on-chain contract is only responsible for sending coins to the winner. In the optimistic case, i.e., when all parties behave honestly, the protocol ends here. Otherwise, if an honest party detects the misbehavior of some other party, the honest one creates a proof of misbehavior and sends it to the on-chain contract. The on-chain contract verifies the claim and punishes the misbehavior.

Next, we state the contribution of this thesis, which leads up to the envisioned scenario.

4.1. Our Contribution

In this thesis, we significantly contribute to the field of privacy-preserving smart contract execution based on MPC, which leads to the scenario above. Our contribution is threefold.

First, we consider the covert security model as a trade-off between security and efficiency. In covert security, malicious parties may deviate from the protocol description, but any cheating attempt is detected with a fixed probability. The idea behind this notion is that the deterrent effect results in malicious parties refraining from cheating at all. We consider an important class of protocols and propose significant efficiency improvements for this class without reducing security in the covert security setting. Our contribution is presented in the following publication, which can be found in Appendix B:

[96] 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**.

Concretely, we consider the offline/online setting, which is a widely used model to increase the concrete efficiency of MPC protocols. In the offline phase, parties compute correlated randomness, which is used to accelerate the online phase. As inputs are only required in the online phase, the offline phase can be computed beforehand, reducing the online phase's response time. Since a large share of the overall complexity is shifted to the offline phase, prior work focused on improving efficiency by considering a covertly secure offline phase while keeping the online phase maliciously secure. We are the first to show that reducing the online phase's security to the covert setting, too, significantly improves overall efficiency. Additionally, we prove that combining a covert offline and a covert online phase provides the same security guarantees as combining a covert offline phase and a malicious online phase. Consequently, we show that reducing the online phase's security to covert security improves overall efficiency without lowering the security. We state the details about this contribution in Section 4.1.1.

Second, we consider covert security with public verifiability. This additional property enables honest parties to transfer knowledge about malicious behavior to any other party, even if this third party was not part of the protocol execution. For this setting, we propose a generic compiler from semi-honest security to covert security with public verifiability in the following publication, which can be found in Appendix C:

[95] 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**.

Prior work mainly focused on the two-party setting of publicly verifiable covert security (PVC). In the multiparty setting, we present the first compiler that supports an arbitrarily high deterrence factor. A property not achieved by prior work in the multiparty setting [77]. A major challenge in realizing PVC is to ensure that honest parties obtain verifiable evidence about malicious parties even if they abort after they are detected. To cope with this challenge, we incorporate time-lock puzzles (TLP), which ensure that parties can decrypt a ciphertext after some time, even without knowing a key. By encrypting the evidence about malicious behavior using TLP, honest parties are able to obtain the evidence even if malicious parties abort. Further details are presented in Section 4.1.2.

Third, we transform PVC protocols to verify proofs of misbehavior inside an on-chain contract efficiently. Our contribution is presented in the following publication, which can be found in Appendix D:

[94] 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**.

Public verifiability is an appealing property to enable verification inside a smart contract. However, prior solutions required expensive computation to verify proofs of misbehavior, making it infeasible for verification inside a smart contract. Our contribution includes modeling the desired privacy-preserving computing scenario outlined above that we capture under the notion of financially backed covert security (FBC). Then, we transform PVC into FBC protocols by reducing the computational requirements for proof-of-misbehavior verification. We incorporate techniques such as a binary search procedure to narrow misbehavior to a single protocol message. We present the details of our contribution in Section 4.1.3.

4.1.1. Improved Covert Online-Offline Protocols

In this section, we present the main contributions of our publication [96], which can be found in Appendix B. In this publication, we consider the covert security model. In covert security, we consider malicious parties that may deviate arbitrarily from the protocol specification. However, honest parties detect any cheating attempt with a fixed probability. The idea of covert security is to deter malicious parties due to the risk of being caught. This is why the detection probability is also called the deterrence factor, often denoted by ϵ. Covert security presents a middle ground between semi-honest and malicious security. In semi-honest security, we consider malicious parties that follow the protocol specification and try to learn additional information from the communication. Therefore, covert security is stronger, as malicious parties may deviate from the protocol specification. In the malicious security setting, corrupted parties may also deviate arbitrarily. However, while there is a fixed probability of successful cheating, i.e., $1 - \epsilon$, in covert security, the successful cheating probability in malicious security is negligible in the security parameter, providing higher security guarantees.

For the covert security setting, we propose significant efficiency improvements for an important class of protocols. Moreover, we show that these efficiency improvements can come for free, as they do not reduce the overall security. In the following, we first explain the class of protocols considered. Then, we explain the traditional approach to achieving covert security and our new approach. Next, we demonstrate the efficiency improvements when applying our new approach. Finally, we state the main idea of our new security composition theorem from which we derive that our efficiency improvements can be obtained without losing security.

Offline/Online Protocols [1]

A widely used technique for constructing efficient MPC protocols is to split the computation into an input-independent offline phase and an input-dependent online phase. This approach aims to shift the bulk of the computational effort to the offline phase so that parties can evaluate the function efficiently once the private

[1]This part is taken verbatim from [96] with slight adjustments.

inputs become available. To this end, the parties precompute correlated randomness during the offline phase. The randomness is then consumed during the online phase to speed up the computation. For the offline phase, the parties use only randomness and no private values as input, and thus, the parties can execute the offline phase already before private inputs become available. Examples of offline/online protocols are SPDZ [78], authenticated garbling [196, 197] and the TinyOT approach [52, 100, 138, 159].

While the offline/online paradigm has traditionally been instantiated in either the honest-but-curious or malicious setting, several recent works have considered how to leverage the offline/online approach to accelerate covertly secure protocols [75, 77]. The standard approach is to take a covertly secure offline phase and combine it with a maliciously secure online phase. Since the offline phase carries a large portion of the overall complexity, this approach significantly improves efficiency. In contrast to the offline phase, we typically rely on a maliciously secure protocol for the online phase. The common belief is that the main efficiency bottleneck is in the offline phase, and thus, there is little value in optimizing the online phase to achieve covert security.

In this thesis, we challenge this belief and show we can significantly improve the overall complexity by using a covertly secure online phase.

Standard Approach to Covert Security

Covert security was introduced by Aumann and Lindell for a two-party garbling protocol [13]. In the two-party garbling setting, one party plays the role of the garbler, and the other acts as the evaluator. The garbler takes a Boolean circuit representation of the function to compute and generates a garbled circuit, an encrypted version of the circuit. Next, the garbler sends the garbled circuit to the evaluator, who evaluates the circuit on a garbled version of the inputs. More concretely, the evaluator obtains the garbled values of its inputs via oblivious transfer (OT) to hide its inputs from the garbler. Additionally, the evaluator receives the garbled inputs of the garbler via a direct message. A maliciously secure OT protocol is used to secure the protocol against a malicious evaluator. The protection against a malicious garbler is more complicated, as the protocol must ensure that the garbled circuit is correct, i.e., represents the desired function.

The covert protocol presented by Aumann and Lindell is based on the *cut-and-choose* approach. This approach builds on a semi-honest protocol and amplifies it to covert security. More concretely, for the two-party garbling protocol, the garbler must generate several garbled circuits and send all of them to the evaluator. Then, the evaluator randomly selects one instance and checks the correctness of all other instances. If all checked instances are correct, the evaluator takes the remaining instance to perform the evaluation. Assuming the garbler generates t garbled circuits, then the evaluator checks $t-1$ of them. Since the checked instances are selected randomly after the generation, the malicious party must decide in which instance to cheat without knowing which instances will be checked. Thus, the probability of detecting cheating is at least $\epsilon \geq \frac{t-1}{t}$.

To check the correctness of a garbled circuit, the evaluator recomputes the garbling algorithm as executed by the garbler. In more detail, after the evaluator randomly selects which instances to check, the garbler reveals all the randomness used while generating the selected instances. Note that the generation of garbled circuits is *input-independent*, i.e., the garbler needs only randomness and no private inputs. Therefore, the garbler can reveal the randomness without breaking privacy. Once the evaluator obtains the randomness for generating the selected instances, it recomputes the garbling process for the desired function and compares the result with the circuits obtained. Note that the garbling process is deterministic if the randomness is fixed. Only if the circuits are identical the check holds.

We can abstract the protocol of Aumann and Lindell [13] into the following blueprint for covertly secure input-independent protocols: (1) execute t instances of a semi-honest protocol, (2) select one random instance, (3) open all other instances by revealing a party's randomness, (4) recompute the opened instance to verify honest behavior, and (5) compute final results via the remaining unopened instance.

We can extend the idea of Aumann and Lindell for a covertly secure two-party garbling protocol to more than two parties and to the input-dependent setting, i.e., where parties use secret data as input. Extending it to the *multiparty setting* poses two challenges. First, a protocol must address the question of selecting the opened instances. The protocol must guarantee that the adversary cannot determine which instances to check, as this would allow the adversary to know in which instance a successful cheating is possible. Since an adversary may corrupt up to all but one party, every party must contribute randomness to the selection process. Second, the verification process might become more complicated. In particular, in a multi-round protocol, one party's behavior depends on the messages received from other parties in previous rounds.

We can solve the former challenge in two ways. On the one hand, the parties can jointly execute a coin-tossing protocol. This protocol outputs a single set that contains all but one instance to open. Then, every party reveals its randomness used in the instances to open. On the other hand, every party can define its own set of opened instances. We call this the watchlist approach, as every party defines its watchlist instead of having a single set of opened instances. This approach comes with a significant drawback. Recall that at least one instance must be unopened so the parties can use this instance to compute the final outputs. To guarantee one unopened instance, the sum of all opened instances must be lower than t. Consequently, every party's watchlist must contain less than $\frac{t}{n}$ instances, which results in a deterrence factor of $\epsilon \leq \frac{1}{n}$. This upper bound is a significant restriction compared to the joint coin-tossing approach.

To allow parties to verify honest behavior in a multiparty protocol instance, every party must reveal its randomness. Additionally, parties send messages to all parties instead of only to the receiver during the execution.[2] Given the randomness of all parties and all messages exchanged during the protocol execution, parties can recompute the entire instance. Parties must get to know all messages during the execution, as otherwise malicious behavior cannot be uniquely attributed to a single party. Consider the scenario where a message of party P_i does not match the recomputed data. It must be the case that either party P_i behaved maliciously or party P_i received an incorrect message from party P_j in the previous round. The verifying party cannot distinguish the two cases if this message from P_j in the previous round is absent. To resolve this problem, parties send messages or hashes of messages to all parties.

We can also extend the cut-and-choose approach to *input-dependent* protocols borrowing techniques from [117] and [140]. Instead of executing several instances of a semi-honest protocol, we execute just a single instance but increase the number of parties. More concretely, every party splits its input into t shares using a t-out-of-t sharing and uses one share as input for one of t simulated parties. Assuming n real parties, the real parties perform a single semi-honest protocol execution between all $t \cdot n$ simulated parties. Afterward, every real party opens $t-1$ simulated parties to show honest behavior during the single protocol execution. Since $t-1$ shares do not reveal anything about the actual input, revealing these shares does not break privacy.

As stated above, the covert security setting aims to provide a middle ground between the efficiency of semi-honestly secure and the security of maliciously secure protocols. When using the cut-and-choose technique, the overhead compared to semi-honest security is around t due to executing t semi-honest protocol instances. Nevertheless, this approach still provides an efficiency benefit compared to malicious protocols if the gap between semi-honest and malicious security is big. This gap is particularly large for malicious protocols that build on the cut-and-choose approach or costly preprocessing protocols [75, 77, 195]. Maliciously secure proto-

[2]If messages must be private during the execution, they can be encrypted under a symmetric key, which the parties generate at the beginning of the protocol execution.

cols based on cut-and-choose require the execution of ρ semi-honest instances [141, 195] for ρ being the statistical security parameters, which is typically set to 40 [98, 195]. Therefore, covert security can provide a trade-off. For instance, executing $t = 10$ semi-honest instances reduces the overhead from 40 to 10, i.e., it saves 75% of the overhead while still giving a 90% chance to detect malicious behavior.

In contrast to the described setting, some protocols provide malicious security at a meager cost compared to semi-honest protocols. Many online protocols incur a small overhead as they benefit from preprocessing or build on very efficient information-theoretically secure techniques. For this class of protocols, covert security based on the cut-and-choose approach offers no advantages.

Our Approach to Covert Security

The overall goal of our contribution in publication [96] is a more efficient combination of offline/online protocols in the covert security setting. As outlined in the previous section, we can use the cut-and-choose approach to achieve a covertly secure offline phase. In contrast, we take a different approach to a covert online phase. On a very high level, we achieve covert security in the online phase by weakening malicious security in contrast to amplifying semi-honest security.

Our idea is based on the observation that by reducing the statistical security parameter of a malicious protocol, we slightly weaken the security but gain efficiency improvements. In addition, this approach avoids the replicated execution overhead of the cut-and-choose approach. We show the usefulness of this approach by applying it to the online phase of the TinyOT protocol [159].

TinyOT is a two-party protocol for computing arbitrary Boolean circuits. The protocol consists of an offline and an online phase. During the precomputation, the parties set up correlated randomness in the form of authenticated bits and authenticated multiplication triples. These values are authenticated in the sense that one party knows the values and message authentication codes (MACs) on these values, while the other party knows MAC keys. During the online phase, the parties compute on these authenticated values to obtain an authenticated output. In the end, the parties verify the MACs to verify the correctness of the computation. Due to the preprocessing of the authenticated values, the online phase is information-theoretically secure and very fast. The security of the online phase is based on the fact that it is hard to guess the MAC key of the other party, where the MAC key is a t-bit string. In our publication [96], we prove that the TinyOT online phase with t-bit MACs and only minor modifications is covertly secure with a deterrence factor of $1 - (\frac{1}{2})^t$. This statement matches our intuition as correctly guessing a t-bit MAC key happens only with a probability of $(\frac{1}{2})^t$ and, thus, the detection probability is $1 - (\frac{1}{2})^t$.

Our following observation is that the MAC size used in the online phase of TinyOT directly impacts the complexity of the precomputation. As mentioned above, the offline phase is responsible for computing authenticated values, i.e., values and corresponding MACs consumed during the online phase. Therefore, by reducing the size of the MACs in the online phase, we also reduce the size of the MACs computed during the offline phase. The MACs are computed using an oblivious transfer (OT) extension protocol. We observed that the MAC size directly corresponds with the number of base OTs performed by the parties. As the execution of base OT requires communication between the parties, fewer base OTs result in lower communication complexity. To summarize, shorter MACs in the online phase reduce the number of base OTs during the preprocessing and, thus, improve the overall efficiency. We can quantify the improvements by calculating the communication complexity of an offline phase computing authenticated values with 40-bit MACs, as required for a maliciously secure online phase, and shorter MACs necessary for a covert online phase. Concretely, we specify a covert offline phase based on the cut-and-choose approach and calculate the communication complexity for a deterrence factor of the online phase up to $\frac{7}{8}$. For this setting, we achieve an improvement of at least 35% compared to the communication complexity required for a malicious online phase.

While we illustrate our new paradigm on the TinyOT protocol, we can apply the approach to other offline/online protocols in the two- and multiparty case, e.g., [52, 100, 138, 196, 197].

So far, we have analyzed the efficiency gains due to a covertly secure online phase. It remains to answer whether or not the security of the overall protocol is reduced by relaxing the online phase to covert security.

Our Security Composition Theorem

In the previous section, we combined a covert offline and a covert online phase. Next, we analyze the security of this combination. Concretely, we are interested in the answer to the question of how is the overall deterrence factor ϵ defined if the protocol consists of a covert offline phase with deterrence factor ϵ_{off} and a covert online phase with deterrence factor ϵ_{on}. While prior work shows that a combination of a covertly offline and a covertly online phase is possible [13], we are the first to answer the question about the resulting deterrence factor explicitly.

Intuitively, combining a covert offline and a covert online phase gives the adversary two opportunities to cheat. Per the definition of covert security, any successful cheating lets the adversary learn all private inputs and gives it complete control over the outputs. We can describe an offline/online protocol as composed of an offline and an online phase. No matter in which phase the adversary tries to cheat, it gains complete control in case of a successful cheating. Consequently, an adversary can always cheat in that phase, where the detection probability is lower, and still gain complete control. Therefore, the intuition tells us that the deterrence factor of the composed protocol should be the minimum of the deterrence factors of the offline and online phases. While the intuition is easy to grasp, we must formally model and prove this composition.

In order to do so, we model offline/online protocols in a hybrid model. The idea behind this model is that an online protocol is described in a setting where the parties have access to an ideal functionality that captures the offline phase. This modeling means that when called by the parties, the offline functionality returns the correlated randomness as computed by the offline phase. Existing composition theorems [13] show that the security of the online protocol holds even if the ideal offline functionality is replaced by a concrete protocol that realizes the functionality. Since we deal with covert security, we extend each ideal functionality with a cheating opportunity for the adversary and a corresponding deterrence factor. Then, the functionality can be replaced by a covert protocol realizing this functionality with the exact same deterrence factor.

To answer the question of how the deterrence factor of the composed protocol is defined when the offline and the online phases are covert, we start with the following setting. Given an online protocol that uses an offline functionality with deterrence factor 1, i.e., no successful cheating is possible, and is covertly secure with deterrence factor ϵ^1_{on}. Then, our composition theorem states that the online protocol is also covertly secure when using an offline functionality with deterrence factor $\epsilon^*_{\mathsf{off}} < 1$. Moreover, the new deterrence factor of the online protocol is defined as $\epsilon^*_{\mathsf{on}} = \min(\epsilon^1_{\mathsf{on}}, \epsilon^*_{\mathsf{off}})$. It follows that the composition theorem formally captures our intuition. For instance, considering a covert offline protocol with a deterrence factor of 50%, we can reduce the deterrence factor of the online protocol to 50%, too, since the overall deterrence factor is the minimum of both.

We conclude our contribution of publication [96] by summarizing that we can improve the overall efficiency of a covert offline/online protocol by reducing the online phase's security to covert, too, without lowering the overall security.

4.1.2. Publicly Verifiable Covert Security

In the previous section, we presented new techniques to improve the efficiency of offline/online protocols in the covert security setting. In this section, we detail our contribution to the publicly verifiable covert (PVC) security setting (cf.

Appendix C). While the covert security setting allows honest parties to detect malicious behavior, parties cannot transfer this knowledge to parties that did not contribute to the protocol execution. Such a feature would increase the deterrent effect since a party's reputation can be damaged publicly. Therefore, Asharov and Orlandi propose the notion of covert security with public verifiability [10]. The notion extends the model of covert security by a third party called a judge, which acts as an adjudicator in case one party blames another. In PVC, the judge is defined as a non-interactive algorithm that obtains a proof of misbehavior from a party and decides whether the claim is valid. If the proof is valid, the accused party is considered malicious. Otherwise, if the proof is invalid, the accusing party is malicious.

Our main contribution in [95] is a generic compiler from semi-honest to PVC security in the multiparty setting. We focus our presentation on input-independent protocols, like the offline protocols described in the previous section, but we can extend our techniques to input-dependent protocols using well-known techniques [117, 140].

Transferring Knowledge About Cheating

The main technical challenge of PVC compared to covert security is to make cheating publicly verifiable. In covert security, the parties can identify malicious behavior. In PVC security, they must also be able to transfer this knowledge to parties that did not contribute during the protocol execution. In order to transfer this knowledge, parties must collect publicly verifiable evidence about cheating during the execution and create a *proof of misbehavior* afterward. Parties can send this proof of misbehavior to any third party who will be convinced about the cheating if the proof is valid.

On a high level, we start with the same blueprint as used for covertly secure protocols based on cut-and-choose. In particular, parties (1) execute several instances of a semi-honestly secure protocol, where all randomness is derived from random seeds, (2) select randomly one instance, (3) open all other protocol executions by revealing the random seeds, (4) recompute the opened instances to verify honest behavior, and (5) continue computation with the remaining unopened instance if all parties behaved honestly. We must solve two main challenges to add public verifiability to the outlined protocol sketch. First, after detecting that a party sent an incorrect message during the execution of one instance, we must allow the accusing party to obtain publicly verifiable evidence about this. In particular, the accusor must convince the judge that the incorrect message originates from the accused party. We can solve the first challenge by requiring each party to sign the protocol transcripts. Since the accused party must have signed the transcript, too, the judge is convinced that the message originates from the accused party if the signature is valid. Second, we must prevent a scenario where a malicious party aborts after learning it is caught before leaving evidence about it. In the covert setting, honest parties can consider such an abort as cheating. However, in the PVC setting, honest parties must also be able to create a proof of misbehavior. As a judge, which is not part of the protocol execution, cannot tell whether the accused party did not send a message or the accusing party did not provide this message, we need to prevent this scenario. We call this feature *prevention of detection-dependent abort*.

We first note that the naive approach of using a coin-tossing protocol does not work, as a rushing adversary can abort after receiving messages from the honest parties but before revealing its own randomness. In contrast, the watchlist approach explained in Section 4.1.1 solves the challenge since the opened instances are selected oblivious. However, as outlined before, this approach has a severe drawback as the deterrence factor is bounded by $1/n$, where n is the number of parties.

We tackle this drawback by proposing a new mechanism to prevent detection-dependent abort. On a high level, we combine the coin-tossing approach with time-lock puzzles (TLP). This combination allows us to construct a compiler for PVC with an arbitrarily high deterrence factor.

Verifiable Time-Lock Puzzle

For constructing a new mechanism against detection-dependent abort, we leverage a cryptographic primitive called time-lock puzzle (TLP) [29, 148, 149, 173]. TLPs provide a means to encrypt messages to the future. The idea behind this primitive is that parties can encrypt a message by creating a TLP. Then, anyone can solve the puzzle even without possessing a secret key. Instead of a key, a party must solve the puzzle, which requires a parameterizable amount of computational work. For instance, the RSW TLP [173] requires computing sequential squaring modulo a composite integer, i.e., for a puzzle $x \in \mathbb{Z}_N$, the solution is $y = x^{2^T} \mod N$. It is conjectured that the solution cannot be obtained faster than performing T sequential squaring operations if the prime factorization of N is unknown. The parameter T is called the hardness parameter, as it defines how much work needs to be done to solve the puzzle.

In order to encrypt a message s, the value x^{2^T} is multiplied by s. Following, a puzzle is the tuple $(x, s \cdot x^{2^T})$. Note that there are two ways to accelerate the generation of a puzzle. On the one hand, if the party knows the factorization of N, then it computes first $\alpha = 2^T \mod \phi(N)$ and next $x^\alpha \mod N$. On the other hand, given a base puzzle of 1, i.e., $p^* = (x, h)$ where $h = x^{2^T}$, a party can generate a puzzle for message s by sampling a random value $r \in [N^2]$ and computing $p = (x^r, s \cdot h^r)$. We base our construction on the second approach to hide the factorization of N from all parties.

A drawback of the RSW TLP is that every party must solve the puzzle independently. The construction provides no means to transfer the knowledge about the solution. To solve this issue, we look at a very related primitive called verifiable delay function (VDF) [37, 164, 199]. In particular, on a construction that is also based on sequential squaring [199]. The primitive allows parties to evaluate a VDF and create a proof that the output results from a VDF evaluation. In this primitive, the evaluation is similar to solving a TLP because it requires computational work, e.g., sequential squaring. We take inspiration from this primitive and integrate the proving technique into TLPs. Our new notion of a verifiable TLP (VTLP) extends a TLP with the capability to transfer knowledge about a solved puzzle. To be concrete, when solving a puzzle p, a party obtains the solution s and a corresponding proof π. Then, s and π can be transferred to a third party, which uses π to verify that s is the solution of puzzle p. The main advantage of this approach is that the third party does not need to solve the puzzle again but can verify a short proof, significantly improving efficiency.

Our publication [95] presents a concrete instantiation of a VTLP based on the RSW TLP. We borrow the proof techniques from the VDF of Wesolowski [199]. On a high level, the proof comprises a succinct public-coin argument made non-interactive using the Fiat-Shamir transformation, i.e., the challenge is computed via a hash function. In more detail, given a puzzle (x, c), where $c = s \cdot x^{2^T}$, the solving algorithm of our VTLP outputs a solution $s = \frac{c}{x^{2^T}}$ and a proof $\pi = x^{\lfloor 2^T/\ell \rfloor}$, where ℓ is the challenge value of the argument. Wesolowski show that π can be computed in $O(T/\log(T))$ multiplications [199]. To verify the solution of a puzzle, the verifier computes the same challenge value ℓ, sets $r = 2^T \mod \ell$, and computes $h' = \pi^\ell x^r$. Then, the solution s is valid if $s = \frac{c}{h'}$, which can be verified without performing the sequential squaring again.

Prevention of Detection-Dependent Abort

Before explaining our new compiler, we provide an intuition of using our new VTLP primitive to prevent detection-dependent abort. The high-level idea is that before learning which instances will be checked, all relevant evidence for cheating is encrypted using a VTLP. Now, even if a malicious party aborts after learning which instances will be checked, the relevant data to detect cheating is already encapsulated in a puzzle. The honest parties can solve the puzzle to obtain the data. This data includes the random seeds used to derive the randomness during the protocol instances. Here, we must pay attention to one crucial subtlety. Recall that the cut-and-choose approach opens all but one instance and keeps the inputs and outputs of the unopened instance confidential. Consequently, the puzzle must not contain all random seeds but only those of the opened instances. We combine the puzzle generation with a coin-tossing protocol to achieve this scenario. We model this via an ideal functionality $\mathcal{F}_{\mathsf{PG}}$, which obtains random coins and the data for the puzzle, including all random seeds. Then, the functionality first computes the unopened instance based on the random coins, discards all data related to this instance, e.g., the random seeds, and finally generates a puzzle containing the remaining data. To prevent parties from cheating during this phase, we require this functionality to be instantiated by a maliciously secure protocol. It seems counterintuitive to require a malicious protocol to achieve PVC since a malicious protocol provides a higher security level. However, we stress that the malicious protocol computes only a specific functionality instead of arbitrary circuits. In particular, the complexity of the malicious protocol is independent of the semi-honest protocol, which we compile to PVC. Consequently, we can compile arbitrarily complex protocols by requiring only a specific functionality to be maliciously secure.

PVC Compiler Description [3]

From a high-level perspective, our compiler works in six phases, which are depicted in Figure 4.1. Initially, all parties jointly execute the seed generation to set up seeds from which the randomness in the semi-honest protocol instances is derived. Since the security of a semi-honest protocol requires the randomness of every party to be uniformly at random, we design the seed generation so that every party contributes randomness to its own and all other parties' seeds. If at least one party is honest, the seed of every party is uniformly random, satisfying the requirement. 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 unchecked instance. The index of this instance is jointly selected randomly in the third phase. Moreover, publicly verifiable evidence is generated so an honest party can blame any malicious behavior afterward. To this end, we first use the puzzle generation functionality $\mathcal{F}_{\mathsf{PG}}$ to generate a time-lock puzzle. Next, each party signs all information required for the other parties to blame this

[3] This part is taken verbatim from [95] with slight adjustments.

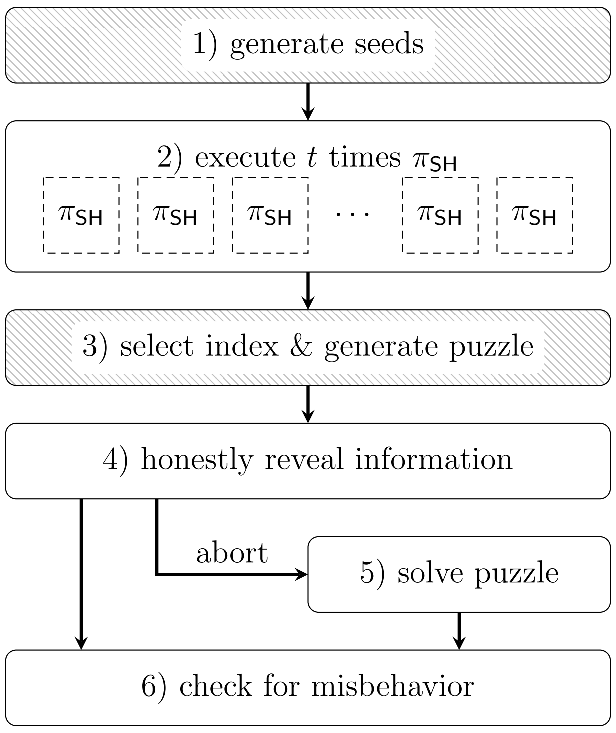

1) generate seeds
2) execute t times π_{SH}
π_{SH} π_{SH} π_{SH} $\cdots$ π_{SH} π_{SH}
3) select index & generate puzzle
4) honestly reveal information
abort
5) solve puzzle
6) check for misbehavior

Figure 4.1.: High-level depiction of our PVC compiler. The rounded boxes illustrate the different phases. A crosshatched phase signals that the phase uses a maliciously secure protocol, and a dashed box denotes a semi-honestly secure protocol.

party. In the fourth phase, the parties either honestly reveal secret information for all but one semi-honest execution or abort. In the 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 must only be executed if a party aborts before revealing the information, we call this the pessimistic case. We stress that no honest party is required to solve a time-lock puzzle if all parties behave honestly. In the sixth phase, parties use the information obtained in phase four or five and check for misbehavior. If a party detects a cheating attempts, the party creates and broadcasts a proof of misbehavior.

A corrupted party may cheat in two different ways in the compiled protocol. Either the party inputs incorrect values into the puzzle generation functionality, or the party misbehaves during executing π_{SH}. The latter means that a party uses randomness different from that 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 $\mathcal{F}_{\mathsf{PG}}$ and can verify that these values are valid. Concretely, the inputs must match the commitments obtained during the initial seed generation. In the pessimistic case, solving the time-lock puzzle reveals the input to $\mathcal{F}_{\mathsf{PG}}$. Note that if parties detect cheating in the optimistic case, they must also solve the time-lock puzzle to generate a publicly verifiable certificate.

If all inputs to $\mathcal{F}_{\mathsf{PG}}$ are valid, 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. If any party misbehaves, an honest party can create a publicly verifiable certificate.

4.1.3. Efficient Verification of Proofs of Misbehavior

In this section, we detail the contribution presented in [94]. Herein, we take the missing steps towards the envisioned scenario outlined at the beginning of Chapter 4. Concretely, our contribution in [94] consists of four components. First, we formally model the envisioned scenario and present a new security notion called financially backed covert security (FBC). Our model includes the parties executing the MPC protocol, a ledger entity responsible for managing coins, and a judging party who controls deposits and adjudicates in case of detected cheating. Second, we present transformations from publicly verifiable covert (PVC) protocols to FBC protocols. Our transformations build on techniques to enable very efficient verification of proofs of misbehavior. Many of these techniques can also be used to improve existing PVC protocols. Third, we present an interactive protocol to facilitate creating and verifying proofs of misbehavior for a new class of protocols. This class comprises protocols without a public transcript. While no PVC protocol without a public transcript is known, we create FBC protocols for this class by leveraging the interactivity of the judge. The interactivity of the judging party is in contrast to the judging algorithm of PVC protocols, where the algorithm is non-interactive per definition. Hence, the interactive property allows us to apply our new FBC notion to this class of protocols, too. Fourth, we implement our judging party as a smart contract for Ethereum. We use our implementation to perform benchmarks and show the practical relevance.

In the following, we highlight the first three parts of our contributions. But first, we recall the envisioned scenario in the light of PVC, as introduced in the last section, and highlight the shortcomings of existing PVC protocols.

Envisioned scenario. We envision a scenario where parties perform private computation, and the computation's output defines a distribution of coins. A vivid example of such a scenario is a poker game, where the outcome of the game defines the party who gets all the coins. Besides a setting where the winner takes it all, more sophisticated applications shall also be supported. To this end, a naive approach is to combine blockchain functionalities with PVC protocols. In more detail, parties deposit coins in a smart contract, perform a PVC protocol on private inputs, and send the computation output to the smart contract. The contract then distributes the coins according to the provided result. If a party detects malicious behavior, it generates a proof of misbehavior and provides the proof to the smart contract. The contract verifies the proof and burns the deposit of the malicious party as a punishment. The combination of smart contracts and a PVC protocol is an appealing candidate for achieving the envisioned scenario.

Shortcomings of PVC protocols. While appealing at first, existing PVC protocols inherit one shortcoming, which renders their use in the envisioned scenario impractical. An important aspect of smart contract execution on a blockchain is that parties must pay fees to execute a smart contract. Every operation is associated with a specific amount of fees; the more operations are executed, the higher the execution fees. When a smart contract must verify a proof of misbehavior, a party must pay fees for all operations required for the verification. Here comes the shortcoming of existing PVC protocols in play, as these protocols require reexecuting an entire protocol instance to compare the recomputed with the signed transcript. Based on this comparison, the contract can decide which party sent the first incorrect message. While parties can perform the reexecution locally, emulating an entire protocol inside a smart contract is prohibitively costly. Therefore, we must design a new mechanism to allow a smart contract to perform a proof-of-misbehavior verification with as few operations as possible.

Before presenting our techniques to achieve an efficient proof-of-misbehavior verification, we provide details on our formal modeling. We capture the modeling under our new notion of financially backed covert security.

Modeling of Financially Backed Covert Security

Our scenario comprises three types of entities. First, parties have secret inputs and jointly perform the private computation. The parties' goal is to distribute coins according to the outcome of the computation. Second, we model the blockchain functionality of maintaining a list of balances via a ledger functionality. Besides storing the balances, the ledger captures the transfer of coins between parties and smart contracts. Third, we formally denote the smart contract as a judge. The judge controls parties' deposits, adjudicates in case of dispute, and distributes the deposits according to the result. The parties interact with the ledger and the judge to achieve their goal of distributing coins based on the outcome of a private computation.

Next, we briefly state the sequence of actions taken by the parties. At the onset, each party deposits coins in the judge smart contract by instructing the ledger functionality to do so. Then, the parties perform the computation and check the honest behavior of all other parties. To allow a party to detect malicious behavior, we require the parties to run a covertly secure protocol. After the covert protocol execution, the parties possess the outputs and forward them to the judge.[4]

We note that our construction must allow honest parties to detect cheating in this step, i.e., if malicious parties send incorrect values to the judge, too. We can achieve this by incorporating simple commitments. We slightly extend the function that is computed by the covert protocol. Instead of outputting the result y to every party, the function first generates n secret shares $(y_1, \ldots, y_n)$ such that $y = y_1 + \ldots + y_n$, and then it generates commitments on these shares, i.e., for $i \in [n]$ $(c_i, d_i) \leftarrow \mathsf{Commit}(y_i)$. The output of the extended function is $(y_i, d_i, \{c_j\}_{j\in[n]})$ for party $P_i, i \in [n]$. Then, the parties exchange signatures on the set of commitments $\{c_j\}_{j\in[n]}$. Eventually, each party sends the set of signed commitments, its share y_i, and its decommitment value d_i to the judge so that everyone can reconstruct the output.[5]

If a party aborts before signing the commitments, no party will learn the reconstructed output, and all parties will consider this behavior an abort. In contrast, if a party signs the commitments, waits to see all other output shares sent to the judge, and refrains from revealing its own output share, this party is considered malicious and punished by the judge. Similarly, if a party provides an incorrect decommitment, then the judge identifies this cheating by trying to open the signed commitments.

If no cheating was detected by all parties during the execution of the protocol, the judge reconstructs the result y and distributes the deposits accordingly. Otherwise, if an honest party detects cheating, a punishment protocol starts. While we allow this protocol to be interactive, i.e., the accusing and the accused parties must interact with the judge, our constructions for the settings with a public transcript

[4]The original motivation in our publication [94] differs from the motivation in this section. Consequently, parties do not send the output values to the judge; thus, the construction in [94] does not contain this step.

[5]Since a party signs the commitments only if it obtained the same as a result of the protocol, it is actually enough if only one party sends the set of signed commitments.

provide a non-interactive protocol. A non-interactive protocol allows the accusing party to send a compact proof of misbehavior to the judge, who can efficiently verify the proof without further interaction. Finally, in the pessimistic case, where at least one party cheats, the judge burns the malicious party's coins and returns the honest parties' deposits. To summarize, in the optimistic case, every party must send two messages to the judge, the deposit and the protocol's result, and the pessimistic case includes a potentially interactive punishment protocol.

Security notion. We formally present a new notion called *financially backed covert* security (FBC), which follows the notion of publicly verifiable covert security and adapts it to our model. An FBC protocol is a tuple $(\pi_{\mathsf{cov}}, \mathsf{Blame}, \pi_{\mathsf{pun}})$, where π_{cov} is a covertly secure protocol, Blame is a non-interactive algorithm that takes the transcript of a covert protocol execution and generates a compact proof of misbehavior, and π_{pun} is a potentially interactive punishment protocol between the parties and the judge. Besides the requirement of π_{cov} being covertly secure, we require $(\pi_{\mathsf{cov}}, \mathsf{Blame}, \pi_{\mathsf{pun}})$ to satisfy *financial accountability* and *financial defamation freeness*. Intuitively, the financial accountability property requires that if an honest party detects cheating, at least one corrupted party loses coins. The financial defamation freeness property protects honest parties by requiring that the probability of an honest party losing coins is negligible. Both properties are the financial analog of PVC's accountability and defamation freeness property, but we are the first to present formal security games, which can be found in publication [94]. Our new notion of FBC allows us to formally analyze the security of our constructions, which we present in the next section.

Constructing Financially Backed Covert Protocols

An FBC protocol must allow honest parties to generate a proof of misbehavior in case cheating is detected. This requirement is similar to PVC protocol, and analogously, FBC protocols require the prevention of detection-dependent abort. In contrast, covert security does not require this property. Therefore, although the FBC security notion requires π_{cov} to be covertly secure only, we start building on PVC protocols to inherit the prevention of detection-dependent abort property.

We further only consider PVC protocols that are built on the cut-and-choose approach. While this requirement seems restrictive, all existing PVC protocols follow the cut-and-choose approach. It is unknown whether PVC protocols can be designed using a different approach. Recall that the cut-and-choose approach builds on the idea of executing several instances and checking all but one of them (cf. Section 4.1.1). Additionally, the parties' random choices during a protocol execution are deterministically derived from a random seed to allow parties to check the honest behavior of other parties.[6] Given the random seed, the party's input, and all incoming messages, the outgoing messages can be deterministically recomputed. More formally, we define a party's initial state as its input and a random seed, i.e., $\mathtt{state}_0 = (x, \mathtt{seed})$. Then, we define a round function round_k which, given the previous state $\mathtt{state}_{k-1}$ and the set of all incoming message $\mathtt{in}_k$, computes a new state $\mathtt{state}_k$ and a set of outgoing messages $\mathtt{out}_k$ of round k, i.e., $(\mathtt{state}_k, \mathtt{out}_k) := \mathsf{round}_k(\mathtt{state}_{k-1}, \mathtt{in}_k)$. We use this abstraction to model the deterministic behavior of every party in each round of each of the several protocol instances executed due to the cut-and-choose approach. During the open phase, the initial states are revealed, which enables the parties to recompute the entire protocol instance.[7] Based on the highlighted observations, we show how to generate compact proofs of misbehavior to facilitate efficient verification. We distinguish between the cases where parties possess a common public transcript and those without a public transcript.

Protocol with public transcript. We know from the previous section on PVC protocols that parties exchange signed protocol transcripts to gather verifiable evidence, but more is needed to verify misbehavior efficiently. Our protocols aim to allow the judge to decide on cheating by recomputing the round function round for only a single party and a single round. This single step means a drastic improvement as the judge must compute the round function for every round and every party in existing PVC protocols. To this end, parties additionally exchange hashes of all intermediate states and sign the set of all state hashes. The idea of this step is to allow honest parties to identify the earliest round function that was computed incorrectly. Then, the judge must recompute only this one, given the previous state and the incoming messages. Since the state hashes and the transcript are signed, the judge is convinced that the accused party computed round based on these inputs. After providing the intuition of our central insight, we take a closer look at the generation of a proof of misbehavior and its verification.

A party uses the blame algorithm after the protocol execution to detect malicious behavior and generate a compact proof of misbehavior. In the setting where parties possess a public transcript, the punishment protocol is non-interactive. Non-interactivity means the protocol consists of only a single message from an

[6]Using a pseudorandom generator (PRG) allows a party to obtain an unlimited amount of pseudorandomness. Hence, the number of random bits is not limited.

[7]In the input-independent setting, where a party's initial state contains only the random seed, we can reveal the initial states of all parties. In the input-dependent setting, we use the technique explained in Section 4.1.1 and open only $t-1$ of t simulated parties per real party.

honest party to the judge, who verifies the proof of misbehavior without further interaction. The generated proof of misbehavior depends on the actual misbehavior of the malicious party, i.e., in which protocol step the malicious party tried to cheat. A malicious party can misbehave in the following steps. First, a party can try to prevent honest parties from learning the initial state. Recall that the initial state can be derived, for instance, from a signed commitment and corresponding signed decommitment. Due to the prevention of detection-dependent abort, a malicious party cannot withhold the decommitment after learning which instances will be checked. It can only provide an incorrect decommitment, which results in a failed opening attempt. Since both values are signed, an honest party can use these values to create a proof of misbehavior. In this case, the judge tries to open the commitment and detects misbehavior if it fails. Second, a party can send an incorrect message during a protocol instance. The blame algorithm emulates the entire protocol execution based on the initial states of all parties and compares the computed transcript with the signed transcript. If the algorithm identifies a discrepancy, it outputs the first incorrect message together with the state and the incoming message of the round function to compute the correct message. Note that the malicious party signs the states and all messages. Thus, the proof of misbehavior contains the signed inputs to the round function and the signed incorrect message. The judge recomputes the specified round function on the provided inputs and compares the result with the signed message. By recomputing a single round function, the judge detects the cheating attempt. Third, a malicious party can modify its internal state during an execution. While state changes due to the round function are valid, other modifications of the state are cheating. Since the parties exchange signed hashes of the intermediate state, the blame algorithm identifies such an invalid state. The proof of misbehavior is similar to the previous case, i.e., it contains the data to allow the recomputation of a single round function and the signed incorrect state. The judge recomputes the round function and compares the new state with the signed one. In all cases, the judge must perform signature verification and only a single round function computation or a single opening of a commitment.

Given a public transcript, our construction supports input-independent and input-dependent protocols. For input-independent protocols, the cut-and-choose approach opens all parties of an opened instance. Opening all parties allows the blame algorithm to recompute the entire protocol transcript. Therefore, it is enough for the parties to exchange message hashes instead of the message in plain. The blame algorithm and the judge compare the hashes of the recomputed message with the signed hashes to detect misbehavior. Depending on the message size, this could reduce the communication complexity. For input-dependent protocols, we cannot open all parties since the private inputs must be kept secret. Therefore, the blame algorithm cannot recompute the entire protocol transcript. Instead, it assumes the message of the unopened parties to be correct. To use messages of unopened simulated parties, the parties must exchange the whole message instead of the hash during the protocol execution. Note that all existing PVC protocols provide one of these variants of a public transcript.

Protocol without public transcript. The construction presented above shrinks the computational work of the judge to the amount of computing a single round of the underlying semi-honest protocol. Additionally, the punishment protocol is non-interactive. We achieve these two properties by providing publicly verifiable evidence about the protocol transcript and the internal states. While both properties are desirable, they come at the cost of increasing communication complexity. Therefore, we present a construction of an FBC protocol that builds on a semi-honestly secure protocol without a public transcript in the following.[8] While doing so, we maintain the low amount of computation required by the judge but require an interactive punishment protocol. Hence, we trade better communication complexity in the optimistic case, where all parties behave honestly, against additional rounds of interaction in the pessimistic case, where the judge adjudicates alleged misbehavior.

We again base our construction on a PVC protocol to inherit the prevention of detection-dependent abort. However, no PVC protocol exists that supports the absence of a public transcript. Moreover, it also needs to be clarified how to construct such a PVC protocol, as the judge algorithm of PVC is restricted to be non-interactive, in contrast to the judge entity of FBC, which is part of an interactive punishment protocol. Nevertheless, we can easily take a PVC protocol with a public transcript, as used in our construction above, and remove the requirement that each party broadcast messages. In this setting, parties still send signed messages to the intended receiver, and for simplicity, we also require that the hashes of internal states are broadcasted (see remark in Footnote 8). To allow parties to detect cheating, we require all parties' initial states to be revealed for an opened instance. This disclosure enables parties to recompute the semi-honest instance and to compare the recomputed message and initial states with the received data. Note that due to the lack of a public transcript, we cannot assume the message from specific parties to be correct and, hence, require that the initial state of all parties be revealed. Consequently, our transformation only works for

[8]In this section, we present the idea of removing a public transcript. The same techniques can be applied to remove the exchange of internal state hashes, further reducing communication complexity. We refer to publication [94] in Appendix D for more details.

input-independent protocols, but not for the input-dependent settings.

Given the initial state of all parties in a semi-honest instance, parties can emulate the protocol execution to identify potential misbehavior. More concretely, parties recompute the protocol based on the revealed initial states and compare the result with the messages and internal state hashes received during the real execution. Note that although the parties do not share a common public transcript, every party receives messages intended for it. While a party cannot check the correctness of a message received by another party, it can check the message with itself as the receiver. If a discrepancy of messages or internal states is detected, the party identifies a potential misbehavior. Note that this is only a potential misbehavior, as the accused party might have received an incorrect message from another party in some previous round. At this point, the checking party cannot be sure who cheated, but it knows that at least some party behaved maliciously. In order to identify the first cheating, every party must check for malicious behavior, and only the accusation of the earliest message in the protocol execution will be considered. To this end, the punishment protocol consists of two phases. During the first phase, every party can submit an accusation. Then, in the second phase, only the earliest accusation of misbehavior is considered. If an honest party makes an accusation, then it can be certain that some malicious party will be punished in the end. It is not mandatory that the accused party is malicious, as this party might have received an incorrect message earlier, but then the accused honest party sends an earlier accusation in the first phase. Note that a malicious party can also send an accusation aiming to create the earliest accusation. However, either the accused party is honest, in which case our protocol guarantees that the honest party can successfully defend against the claim and the accusing party is considered malicious, or the accused party is also malicious, in which case one of the malicious parties is punished.

After identifying the earliest accusation, the second phase of the punishment protocol adjudicates the accusation. In this phase, the judge must decide if the accused party behaved incorrectly during the protocol execution or if the accusation is unfounded. We consider the case where the accuser claims that the accused party sent an incorrect message in round k (the case of a faulty internal state is analogous). Here, the major challenge is that the accuser cannot provide verifiable data to the judge to recompute $\mathtt{round}_k(\mathtt{state}_{k-1}, \mathtt{in}_k)$ and resolve the dispute. In particular, the accuser has no signature on the set of incoming messages $\mathtt{in}_k$ of the accused party. Nevertheless, both parties have an entire history of messages that were recomputed during the identification of misbehavior. The accuser sends its history of messages up to round k to the judge, and the accused party must respond whether it agrees with the history. We distinguish both cases. If the accused party agrees, then the judge uses the provided messages to recompute round_k. This mechanism allows the judge to settle the dispute immediately, the same way as in our previous construction. In the other case, where the accused party disagrees with the suggested history, they perform a binary search to identify the first message they disagree with. Since this procedure is an interactive protocol, we call it a *bisection protocol* [56]. During this subprotocol, the history is divided into two halves in each round to narrow down the disagreement to a single message. Note that all messages are sent to the judge. However, the judge must only perform simple operations during the step, e.g., adjusting the search window and maintaining timeout intervals. If any party does not respond in time, this party is considered malicious and the punishment protocol stops. After the bisection subprotocol is finished, the parties identify the first message they disagree on. Per definition, they have agreed on all the messages before. Let the message in doubt be an outgoing message of party P_j in round k' (P_j can be any party of the protocol). Then, given all incoming messages, on which both parties agreed, and the internal state, the judge can recompute $\mathsf{round}_{k'}$ of party P_j. Recall that the parties exchanged signatures on all internal state hashes after the execution. Thus, due to its recomputation, the accuser can provide the internal state matching the hash. By computing the single round function $\mathsf{round}_{k'}$, the judge can check which party provided an incorrect message history and punish the malicious party.

In total, we trade additional interaction in the punishment protocol against less communication during the execution of the semi-honest protocol instances. Due to the lack of a public transcript, the accusing and the accused parties first need to agree on a message history. Then, the judge can resolve the dispute by recomputing a single protocol step. The amount of computation required by the judge to solve the dispute is basically the same as in the setting with a public transcript.

4.2. Related Work

At the beginning of this chapter, we provide an overview of different approaches for privacy-preserving smart contract execution. Moreover, we present references for solutions based on trusted execution environments, zero-knowledge proofs, and fully homomorphic encryption. Therefore, we focus on MPC-based solutions in the following. Additionally, we give an overview of MPC protocols related to our techniques, i.e., covert and publicly verifiable covert security and MPC with financial instruments.

Private smart contracts via MPC. Enigma [215] achieves private smart contract execution, reduced on-chain storage cost, and better scalability via MPC. To this end, Enigma combines an existing blockchain with an off-chain network consisting of MPC nodes. These MPC nodes are responsible for carrying out private computation, maintaining a distributed database, and taking over expensive computations. Enigma uses publicly auditable SPDZ [20] to perform computation. Auditable MPC is an extension of maliciously secure MPC, i.e., privacy and correctness holds against malicious adversaries. This is in contrast to our solutions, which consider covert adversaries. The idea of auditable MPC is to create an audit trail during the computation so that a third party can verify the correctness of the result. The verification of correctness is also possible even if all computing parties are corrupted. However, auditable MPC also requires at least one honest party for guaranteeing privacy. The verification process of auditable MPC requires recomputing the entire circuit, which is prohibitively expensive for automatic verification by a smart contract. In contrast, our constructions specifically facilitate verification of proofs of misbehavior via a smart contract. To this end, our solution leverages the one-honest-party assumption that is anyway mandatory for our construction and auditable MPC to ensure privacy.

Hawk [132] provides a compiler that turns smart contracts into cryptographic protocols with additional privacy guarantees. However, Hawk relies on a centralized facilitating manager. Although the manager cannot break correctness due to the use of zero-knowledge proofs, it receives the private inputs of the users and, thus, must be trusted to ensure privacy. While [132] states that the manager can be equipped with a trusted execution environment (TEE) to remove this trust assumption, incorporating TEEs adds strong hardware assumptions. The follow-up works zkHawk [16] and V-zkHawk [17] replace the trusted manager with an MPC protocol. Both works optimized the operations executed within the MPC protocol to make the computation more efficient. [16] mentioned that the maliciously secure general-purpose SPDZ protocol [7, 78] can be used to instantiate the MPC protocol. We deem our techniques for covert and publicly verifiable covert security also applicable to realize the MPC protocol with improved efficiency.

The Eagle protocol [19] combines outsourced MPC [118] with insured MPC [21] to achieve privacy-preserving smart contract execution. Outsourced MPC facilitates a setting where clients provide inputs to a set of servers that perform the computation. This setting enables lightweight clients or clients to go offline during the computation. Eagle motivates the use of outsourced MPC via the latter argument. Insured MPC [21] provides fair output delivery with financial penalties. If a corrupted party aborts during the evaluation of the private smart contract, it will be financially punished. The punishment is achieved by allowing honest parties to post a challenge on-chain to start a timeout by when the corrupted party must respond. Note that Insured MPC builds on a maliciously secure MPC protocol. Therefore, any cheating attempt of corrupted servers is detected except with negligible probability, and thus, an incorrect message can be considered as an abort. The main difference between Eagle and our solutions is the considered adversary setting. While [19] builds on protocols that consider malicious adversaries, we focus on the covert security setting. Additionally, Eagle considers a setting where clients provide inputs to a set of servers who perform the execution. In our work, we take the scenario where the parties that posses private inputs also perform the computation.

Covert security. Covert security was initially studied by Aumann and Lindell [13]. [13] presents a two-party protocol based on garbled circuits [207]. The protocol builds on a semi-honestly secure protocol and uses the cut-and-choose approach to enable honest parties to detect malicious behavior. Goyal et al. [107] presents the first multiparty covertly secure protocol. [107] is also based on the garbling techniques adapted to the multiparty setting using techniques from Beaver et al.[24]. Additionally, Goyal et al. also use the cut-and-choose technique. Subsequently, the covert security model was considered by a line of work, including [110, 111, 165]. All these works are based on the cut-and-choose approach and either improve the general-purpose covertly secure protocols or apply the covert security model to specific functionalities, e.g., private set intersection [111], oblivious polynomial evaluation [110]. Additionally, implementations of covertly secure protocols are provided (e.g., [165]).

Damgård et al. presented the first generic compiler from semi-honest to covert security [73]. Their compiler works for the honest majority setting and achieves a fixed deterrence factor of 1/4. Lindell et al. [140] combined the famous IPS compiler [117] and the covert security setting. On the one hand, [140] uses a covertly secure protocol as the inner protocol, thus achieving better asymptotic complexity. On the other hand, Lindell et al. show modifications of the IPS compiler to achieve covert security in the dishonest majority setting from an information-theoretically secure honest majority outer protocol and a semi-honestly secure dishonest majority inner protocol.

A range of research papers further considered the covert security setting. Damgård et al. [74] and Damgård et al. [75] adapted the maliciously secure SPDZ [78] protocol to the covert security setting and provided implementations. Kamara et al. [123] used the covert setting in realizing efficient server-aided secure function evaluation. Zeng et al. [210] propose a framework for oblivious transfer in the covert security setting.

Lindell first realized that a range of maliciously secure protocols based on the cut-and-choose approach achieve covert security by reducing the statistical security parameter [139]. [139] also presents an optimized two-party garbling protocol for malicious and covert security.

Mohassel and Riva [156] introduced a new security notion, a strengthened variant of covert security. In more detail, they combine covert security with the idea of input-dependent abort [115] or limited leakage of [113, 155]. In their notion, called covert security with input-dependent abort, correctness always holds, and only privacy breaks with probability $1 - \epsilon$, where ϵ is the deterrence factor. Moreover, a privacy breach leaks only a single bit to the adversary.

The following works also combined the covert security setting with other security models. Kolesnikov et al. [130] combined the dual-execution model of [113, 155] with covert security to achieve a "continuum of cost-security trade-offs" [130] in the two-party setting. Küpçü and Mohassel [136] combined a covertly secure two-party protocol with a trusted arbiter to add fairness guarantees.

Asharov et al. [9] present OT extension protocols in the active and covert security setting.

Choudhouri et al. [64] analyzed the round complexity of covertly secure protocols. Moreover, they compared the round complexity of these protocols depending on the deterrence factor.

Publicly verifiable covert security. The publicly verifiable covert security (PVC) setting was introduced by Asharov and Orlandi [10]. The idea behind this notion is to extend covert security with a mechanism allowing honest parties to transfer knowledge about cheating to third parties. Even if these parties are not present during the computation, they can be convinced about misbehavior. [10] introduced the notion and presented a two-party protocol satisfying PVC security. Kolesnikov and Malozemoff [129] and Hong et al. [112] proposed improvements to the two-party protocol of [10].

Damgård et al. [77] presented the first black-box compiler from semi-honest to PVC security. [77] focused on the two-party setting and presented an informal extension to the multi-party setting. The drawback of their multiparty compiler is a limited deterrence factor of $1/n$, where n is the number of parties, or a higher deterrence factor at the cost of multiple sequential protocol executions.

Our compiler presented in Section 4.1.2 is the first fully formalized generic multiparty compiler from semi-honest to PVC security. The primary novel technique to achieve PVC in the multiparty setting is using time-lock puzzles (TLP). Concurrent and independent of our work, Scholl et al. [176] also proposed a multiparty compiler for PVC, and Scholl et al. incorporated TLP, too. The main difference is the creating and solving of the TLPs. While our compiler computes one TLP for all parties inside an actively secure MPC protocol, [176] requires every party to create one puzzle, resulting in n different puzzles. Note that [176] also requires an actively secure MPC protocol, although no TLP is created inside the circuit. In both compilers, the circuit size computed by the actively secure protocol is independent of the semi-honest protocol size. As a result of the different puzzle creations, the parties in our protocol must solve only one puzzle, while the parties in their construction must solve all n puzzles. See our publication [95] in Appendix C for a more detailed comparison.

Attema et al. [11] presented a generic compiler transforming semi-honest to PVC protocol in the honest majority setting. Their compiler follows the same blueprint as ours and [176] but avoids TLP due to the honest majority setting.

Liu et al. [144] presented the first constant-round protocol for private function evaluation (PFE) with PVC security. In the PFE setting, one party possesses a function f, the other or both parties possess private inputs x_A and x_B, and the parties learn at most the output $f(x_A, x_B)$ but nothing beyond. In particular, the other party does not learn the function f and the first party's input.

Similar to the combination of dual execution and covert security as proposed by Kolesnikov et al. [130], Liu et al. [143] combined the dual execution model [113, 155] with PVC. This combination strengthens PVC, as the adversary learns only 1 bit of information in case of successful cheating.

Duan et al. [86] presented the notion of security against an honorific adversary. This type of adversary may deviate from the protocol description but fear to damage its reputation publicly. Therefore, in a protocol where cheating attempts may be detected, and evidence about the attempts are present, this type of adversary refrains from cheating. Duan et al. introduced an independent party called an auditor, which assists in detecting cheating. In contrast, the publicly verifiable covert security setting [10] also allows the transfer of knowledge about cheating to third parties but does not require an auditing party.

MPC with financial mechanism. As stated above, insured MPC [21] combined MPC with distributed ledgers. However, the motivation behind this work is to realize fair output delivery, i.e., corrupted parties that learn the output and prevent honest parties from knowing the output are financially punished.

Besides [21], the idea of combining MPC with cryptocurrencies to integrate financial penalties for aborting parties was considered by a line of work (e.g., [5, 6, 26, 133, 134, 135]). While these protocols combine MPC and blockchain as we do, their focus is on achieving a variant of fairness, and we focus on attaining private smart contract execution.

Similar to our work presented in Section 4.1.3, Zhu et al. [213] combined covert security with financial punishments. They present a two-party garbling protocol and achieve an efficient verification of proofs of misbehavior via a smart contract. Since their protocol is tailored to the two-party garbling setting, the judge only needs to verify the garbling process, which is a non-interactive algorithm. In contrast, our work [94] applies to multiparty protocols and solves challenges that come up when the judge must verify interactive protocols.

5. Decentralized Issuing of Anonymous Credentials

Anonymous credentials provide cryptographic means to increase the privacy of individuals and, thus, play an important role in combination with blockchain technology in the realm of self-sovereign identities (SSI). Such a credential scheme allows an issuer to generate credentials for users, who then can use the credentials to provide verifiable claims to third parties, e.g., to prove authorized access to a web service. During this process, anonymous credentials provide two essential properties. First, the unlinkability property guarantees that verifiers cannot link two disclosures of credentials of the same identity. Second, the selective disclosure property enables users to reveal only parts of their credentials, i.e., specific attributes, instead of all information. Both properties provide important privacy features for digital identities. The selective disclosure property makes anonymous credentials particularly suitable for SSI, as it equips the user with the power to control the amount of data revealed. Thus, anonymous credentials are an attractive building block towards putting control over identities to the users instead of the service providers as envisioned by the concept of SSI. Besides anonymous credentials, blockchain technology is an appealing building block for SSI due to its decentralized nature. More concretely, blockchains provide decentralized and tamper-proof storage for public metadata like decentralized identifiers, information about the scheme used to create the credentials, and revocation lists. The decentralized nature matches the goal of SSI, which is to remove the data and trust from centralized service providers. The combination of anonymous credentials and blockchain technology is demonstrated by the Hyperledger Indy project [114].

One of the most prominent solutions for anonymous credentials is the BBS+ signature scheme [12, 187]. Abstractly speaking, an issuer generates a credential by blindly signing a set of attributes. Then, the signature holder can use zero-knowledge proofs to prove the possession of a valid signature on specific attributes. The suitability of BBS+ stems from several appealing features. First, BBS+ allows the issuer to create a signature on a large set of attributes while keeping the credentials' size constant. Second, there exist efficient protocols for creating blind signatures [12, 187] and, third, the existence of efficient zero-knowledge proofs that support the selective disclosure of attributes [12]. The prominence of BBS+ is evident by recent research [85, 187], industrial implementations [150, 153, 191], standardization efforts by the W3C Verifiable Credentials Group and IETF [28, 146], and the usage in further applications [12, 48, 49, 53, 61].

We briefly recall the details of the BBS+ signature scheme and refer to the Preliminaries 2.5 for a full formal definition. A signature on a set of message $\{m_\ell\}_{\ell \in [k]}$ is a tuple (A, e, s), where e, s are random nonces from $\mathbb{Z}_p$ for prime p and $A = (g_1 \cdot h_0^s \cdot \Pi_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$ for x being the secret key. $g_1 \in \mathbb{G}_1$ is a generator and $\{h_\ell\}_{\ell \in \{0,\dots,k\}}$ are random elements in $\mathbb{G}_1$. To verify a signature under a public key g_2^x, the verifier uses a bilinear map $\mathsf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ and checks if $\mathsf{e}(A, g_2^x \cdot g_2^e) = \mathsf{e}(g_1 \cdot h_0^s \cdot \Pi_{\ell \in [k]} h_\ell^{m_\ell}, g_2)$ for g_2 being a generator of $\mathbb{G}_2$.

As defined above, the scheme is designed for a single signer, which poses the risk of a single point of failure. Concretely, a single party possessing the secret key x constitutes an attractive target for an adversary. When the adversary successfully corrupts the signer, it has complete control over the secret key. Such corruption is a severe scenario, allowing the adversary to create valid credentials for arbitrary attributes. Moreover, the maliciously generated credentials look like honestly created ones from the verifiers' perspective.

To mitigate such a single point of failure, we can resort to multiparty computation (MPC) to realize the signing task. More concretely, instead of allowing a single party to create credentials, we split the secret key x into a set of n shares, i.e., $x_1, \dots, x_n$, distribute the shares to n different parties, and require a subset of all parties to jointly create credentials. The size t of the minimum number of required signers is called the threshold, and the concept of distributing a cryptographic task between multiple parties is called *threshold cryptography.* The critical aspect of threshold cryptography is that no subset of less than t parties can perform the cryptographic task at hand, i.e., creating valid credentials in our use case. Consequently, instead of corrupting only a single signer as in the traditional setting, the adversary must corrupt t parties in the threshold setting. To realize the signing algorithm of BBS+, we envision an MPC protocol tailored to this cryptographic task. The advantage of using a tailored protocol instead of general-purpose MPC is improved efficiency since the protocol can exploit the specific structure of the problem.

5.1. Our Contribution

In this thesis, we aim to apply the decentralization paradigm of blockchain technology to new components paired with blockchains. To this end, we distribute the issuance of anonymous credentials, which are combined with blockchains in the context of SSI. We present our contribution in the following publication, which can be found in Appendix E.

[93] S. Faust, C. Hazay, D. Kretzler, L. Rometsch, and B. Schlosser. *Non-Interactive Threshold BBS+ From Pseudorandom Correlations.* Cryptology ePrint Archive, Paper 2023/1076. `https://eprint.iacr.org/2023/1076`. 2023. **Part of this thesis**.

Concretely, we present a t-out-of-n threshold BBS+ signing protocol, which allows the issuing of anonymous credentials in a distributed manner. We build our construction on the offline/online model, which we have already seen in Chapter 4. While the offline phase is input-independent and, thus, provides a preprocessing phase, the online phase comprises the actions taken after the message to be signed is available. Designing an efficient online phase enables fast response time upon a signature request. In constructing our offline phase, we use a recently introduced trend to build precomputation material based on pseudorandom correlations. The advantage of these correlations stems from the existence of primitives that allow the setup of correlations with sublinear communication complexity in the number of correlated tuples. Then, we leverage the precomputation material to design a non-interactive online phase, i.e., the signers respond to a signature request without interaction among themselves. Our scheme is the first threshold signature scheme to simultaneously achieve sublinear communication complexity in the offline phase and non-interactivity during the online phase.

In the remaining part of this section, we first highlight the challenges of thresholdizing the BBS+ signing algorithm. Then, we present our main results in a top-down approach, i.e., we start presenting our online phase, which builds on precomputation material, and then explain how this material is computed in the offline phase.

5.1.1. Challenges of Thresholdizing the BBS+ Signing Algorithm

To thresholdize a cryptographic task, we first secret share the secret information, e.g., the secret key x. Typically, we use additive secret sharing or Shamir's secret sharing [178]. The first one works by sampling n random elements under the constraint that $\sum_{i\in[n]} x_i = x$ and is suitable for the n-out-of-n case, i.e., all n parties must contribute to perform the cryptographic task. In contrast, Shamir's secret sharing enables fixing an arbitrary $1 \leq t \leq n$ for a t-out-of-n access structure. We present our ideas for additive sharing for simplicity, but our construction, as presented in [93], also supports a flexible t. Both types of sharing allow the share-holders to perform linear operations on their shares locally. For instance, let $[a]$ and $[b]$ denote secret shared values a and b, then the parties can compute a secret sharing of $[a+b] = [a]+[b]$ by locally adding up their shares, i.e., party P_i computes $a_i + b_i$. In contrast to linear operations, non-linear operations like multiplications require interaction between the parties and, hence, are comparatively way more expensive than linear operations. Therefore, thresholdizing cryptographic tasks with non-linear operations is much more complicated than distributing linear operations.

The BBS+ signing algorithm, as introduced at the beginning of this section and formally stated in 2.5, contains one non-linear operation in computing the inverse of $x + e$, where x is the secret key and e being a random nonce. More concretely, the inverse is within the exponent, i.e., for some element $M \in \mathbb{G}_1$, we must compute $M^{\frac{1}{y}}$, where $y = x + e$. Note that y is secret-shared between n parties, as x is secret-shared, and e must be secret-shared, too, to prevent a corrupted party from biasing the randomness. To compute $M^{\frac{1}{y}}$ without revealing the value y in clear, Bar-Ilan and Beaver [18] present a simple but elegant way. First, the parties separately open the value $B = M^a$ and $\delta = a \cdot y$ for a random secret shared value a, where opening means that each party sends its share to every other party. Then, each party can locally compute the inverse of δ to compute the final result as $B^{\frac{1}{\delta}} = (M^a)^{\frac{1}{a \cdot y}} = M^{\frac{a}{a \cdot y}} = M^{\frac{1}{y}}$. Note that revealing the value δ does not break the privacy of y since it is perfectly hidden by the random value a. However, the computation of $\delta = a \cdot y$ is still a non-linear operation requiring interaction between the parties. An important observation is that this operation only depends on the random values a and e and the secret key x. In particular, the calculation is independent of the message to sign. This fact allows us to move the computation in the offline phase and, hence, remove the expensive non-linear operation from the online phase. As a result, we show how to construct a non-interactive online phase in the following section.

5.1.2. Building a Non-Interactive Online Phase

Requirements on the preprocessing. Now, we show how to design a non-interactive online phase from specific correlations. In the next section, we explain how to compute the required correlations as part of the offline phase from efficiently generatable correlations. First, we recall the BBS+ signing algorithm. A signature on a set of messages $\{m_\ell\}_{\ell \in [k]}$ is a three-tuple (A, e, s), where $e, s \in \mathbb{Z}_p$ are two random nonces and $A = (g_0 \cdot h_0^s \cdot \Pi_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$. For computing such a signature, we consider the secret key x and the nonces e, s to be secret-shared, i.e., signer P_i possesses x_i, e_i, s_i such that $\sum_{i\in[n]} x_i = x$ and the same holds for e and s. Additionally, to apply the trick of the Bar-Ilan and Beaver inversion protocol [18], we require a secret-shared random value a, i.e., P_i possesses a_i. Next, we observe that we can compute A as the product of $M_1^{\frac{1}{x+e}}$ and $M_2^{\frac{s}{x+e}}$, where $M_1 = (g_1 \cdot \Pi_{\ell\in[k]} h_\ell^{m_\ell})$ and $M_2 = h_0$. Note that for M_1, we only have the inversion of $x+e$ in the exponent, and for M_2, we must compute $s \cdot (x+e)^{-1}$. To apply the inversion trick to both cases, we define two secret-shared masked values δ and α, where $\delta = a(x+e)$ and $\alpha = a \cdot s$. We can use the same random mask a in both terms since s and $x+e$ are independent and random due to the randomness of e and s. Summarizing, we require the precomputation to provide to party P_i for $(i \in [n])$ a tuple $(a_i, e_i, s_i, \delta_i, \alpha_i) \in \mathbb{Z}_p^5$ which satisfy the following correlations

$$\begin{gathered}\delta = \sum_{i\in[n]} \delta_i = a(x+e), \qquad \alpha = \sum_{i\in[n]} \alpha_i = as \\ \text{for } a = \sum_{i\in[n]} a_i, \qquad e = \sum_{i\in[n]} e_i, \qquad s = \sum_{i\in[n]} s_i.\end{gathered} \tag{5.1}$$

Note that all correlations are independent of the message and, thus, can be computed in the offline phase. We call the tuple $(a_i, e_i, s_i, \delta_i, \alpha_i)$ a presignature. Next, we describe the actions required to compute a valid signature without interaction between the signers given these presignatures.

From presignatures to signatures. The online phase of our distributed signing protocol is non-interactive because the signers do not interact with each other. We consider a client-server setting where a client requests a signature from a set of signers. This setting matches the blind signing required for anonymous credentials. While we present the non-blind signing here, the protocol can be extended straightforwardly. We refer to our publication [93] for further details. The signing request sent by the client includes the set of messages to be signed, i.e., $\{m_\ell\}_{\ell\in[k]}$. Then, each signer uses a presignature to respond with a partial signature. Finally, the client aggregates all partial signatures to a valid signature.

To create a partial signature, a signer P_i first computes $M_1' = (g_1 \cdot \Pi_{\ell\in[k]} h_\ell^{m_\ell})^{a_i}$ and $M_2' = h_0^{\alpha_i}$. Then, a partial signature is the tuple $(A_i, \delta_i, e_i, s_i)$, where $A_i = M_1' \cdot M_2'$ and $(a_i, e_i, s_i, \delta_i, \alpha_i)$ is the consumed presignature. Note that the secret key is included in the partial signature via the shares δ_i, a sharing of $a(x+e)$. The idea behind this partial signature is that the client locally inverts $\delta = \sum_{i\in[n]} \delta_i = a(x+e)$, analogously to the Bar-Ilan and Beaver trick. By taking the product of all A_i's to the power of δ^{-1}, the a value vanishes. Since we took h_0 to the power of α, the s value remains. More concretely, a client reconstructs δ, e, and s by summing up the shares and then computing $A = (\Pi_{i \in [n]} A_i)^{\frac{1}{\delta}} = ((g_1 \cdot \Pi_{\ell \in [k]} h_\ell^{m_\ell})^a \cdot h_0^{as})^{\frac{1}{a(x+e)}} = (g_1 \cdot h_0^s \cdot \Pi_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$. This calculation yields a valid signature (A, e, s). Since the signature has the same form as in the single-signer setting, we can use the same verification algorithm. We finally stress the non-interactivity of our setting, as the signers do not need to communicate with each other in the online phase. There is only a request by the client followed by responses from the signers. This pattern is the minimal interaction required by a client-server setting.

We achieve the non-interactive online phase by requiring a tailored set of correlations (cf. Equation (5.1)). In the next section, we show how the signers set up these correlations during the offline phase.

5.1.3. Constructing an Offline Phase from Pseudorandom Correlation

Computing presignatures from cross-products. The offline phase aims to provide presignatures as defined in Equation (5.1) to every signer. We first observe that the secret sharings of e, s, and a can be defined by every signer locally sampling its share, e.g., P_i samples e_i, and then we define e, s, and a as the sum of the shares, e.g., $e = \sum_{i \in [n]} e_i$. This sampling is straightforward and requires no interaction between the signers. In contrast, we require interaction to set up secret sharings of $\delta = a(x + e)$ and $\alpha = a \cdot s$. Recall that x is the secret-shared secret key. We observe that

$$\begin{aligned} \alpha &= a \cdot s \\ &= (a_1 + \ldots + a_n) \cdot (s_1 + \ldots + s_n) \\ &= a_1 \cdot s_1 + \ldots + a_1 \cdot s_n + \ldots + a_n \cdot s_1 + \ldots + a_n \cdot s_n \,, \end{aligned}$$

i.e., we can express the product of two secret-shared values as the sum of multiplications of two secret shares. While every party P_i can locally compute $a_i \cdot s_i$, parties P_i and P_j for $j \neq i$ must interact to compute $a_i \cdot s_j$ and $a_j \cdot s_i$. We call the multiplications of two secret shares of different parties the cross-products or cross-terms. The same observation holds for $\delta = a(x + e) = a \cdot x + a \cdot e$ since δ consists of two multiplications of secret-shared values. One approach to compute the cross-terms is to use a multiplication protocol or a multiplicative-to-additive share conversion protocol (e.g., [23, 83, 105], and references within [14]). The idea of these protocols is to compute the multiplication and return additive shares. Given these additive shares, every party can locally sum up the additive shares of all cross-products to obtain an additive sharing of α and δ.

Multiplicative-to-additive share conversions for the product of two random values can also be expressed via oblivious linear evaluations (OLEs). Intuitively, OLEs are two-party correlations providing (x, y) to one party and (v, w) to the other, where x, v and w are random elements and $y = v \cdot x + w$. By rearranging the equation, we get $y - w = v \cdot x$, i.e., y and w are additive shares of the multiplication $v \cdot x$. Since the values of the cross-products are random shares, we can use OLE to transform the cross-products into additive shares. More concretely, each pair of party (P_i, P_j) requires an OLE correlation for transforming each of the cross-products $a_i s_j$, $a_j s_i$, $a_i e_j$, and $a_j e_i$. The cross-products $a_i x_j$ and $a_j x_i$ are exceptional cases since the secret key shares are fixed for all presignatures, while e, s, and a are fresh random values each time. To fix the secret key shares for all correlation outputs, we utilize vector OLE (VOLE). VOLE is similar to OLE but outputs $(x, \vec{y})$ to P_1 and $(\vec{v}, \vec{w})$ to P_2 with $\vec{y} = \vec{v} \cdot x + \vec{w}$, i.e., $\vec{v}, \vec{w}$, and $\vec{y}$ are vectors while x is a constant scalar. Hence, a VOLE correlation provides many correlations $y_k = v_k \cdot x + w_k$ for $k \in |\vec{y}|$, where x is fixed.

Computing cross-products in sublinear communication. Computing multiplications of additively shared values is possible via multiplication protocols (e.g. [83, 105]) in linear communication complexity. In contrast, Boyle et al. [42, 46, 47] show how to generate OLE and VOLE correlations, which we have seen to be analogous to multiplicative-to-additive share conversions for the product of two random values, in sublinear communication complexity. They introduce a new cryptographic primitive called pseudorandom correlation generator (PCG) and present constructions to generate pseudorandom OLE and VOLE correlations under variants of the learning parity with noise assumption. Due to the advantage of PCGs over multiplication protocol in terms of communication complexity, we build our precomputation phase on this primitive. In more detail, a PCG consists of two algorithms, Gen and Expand, and realizes a correlation, e.g., OLE or VOLE. The Gen algorithm takes as input a security parameter and potential input parameters, which depend on the realized correlation. For instance, the VOLE correlation fixes a scalar, provided as input to the algorithm. Given the inputs, Gen generates two short seeds, which are distributed to two parties. Then, each party locally evaluates the expansion algorithm Expand on its seed. This algorithm outputs a large batch of tuples that satisfy the correlation. For instance, considering a PCG for the VOLE correlation, the first party computes $(x, \vec{y}) = \mathsf{Expand}(\mathsf{seed}_0)$ and the second one obtains $(\vec{v}, \vec{w}) = \mathsf{Expand}(\mathsf{seed}_1)$. Note that each party can locally evaluate the expansion algorithm without interaction with the other party. In contrast, the seed generation algorithm is either executed by a trusted dealer or via a distributed protocol, depending on the use case. Even in the distributed setting, the communication is sublinear in the number of obtained correlations since the parties must set up short seeds only.

Security-wise, PCGs provide two properties. Intuitively, the pseudorandom $\mathcal{Y}$-correlated output property states that the outputs of the expansion algorithm look pseudorandom under the constraint that they satisfy the correlation $\mathcal{Y}$ and the security property states the no party can learn the output of the other party even when knowing its own seed. Due to the benefit of requiring only sublinear communication, PCGs are an appealing candidate for realizing the OLE and VOLE correlations required by our offline phase. However, when using this primitive, we encounter some challenges.

Black-box use of pseudorandom correlation generators. Intuitively, the challenges arise from using PCGs in a multiparty setting where we consider a malicious adversary. Prior work presents definitions only for passive adversaries [46, 47]. First, we must allow parties to obtain the same outputs in several PCG instances in the multiparty setting. To illustrate this feature, recall that we use the PCGs, for instance, to compute the cross-products of as, i.e., $a_i \cdot s_j$, $a_i \cdot s_k$, and so on for i, j, k being different values. In this scenario, we require the a_i value to be the same in both cross-products. Therefore, party P_i must obtain the same in both PCG instances corresponding to these cross-products. We account for this feature by adding a *programmability property* to the definition of PCGs. This extension follows the definition of [46, 47] and allows parties to specify input parameters to the seed generation. Then, the outputs of the expansion algorithms are deterministically derived from these parameters. This derivation allows the output to be programmed in several instances. Next, we account for a malicious adversary by giving the adversary the power to define its input parameters. This setting is in contrast to prior definitions, which sample the inputs of the adversary at random and, thus, consider the passive adversary setting. Finally, we identify a missing piece in the definitional framework when using PCGs in a black-box way in our offline phase. We add a *key indistinguishability property* stating that the adversary cannot learn any information about the other party's input parameters, even knowing its own seed. This property is necessary to treat PCGs in a black-box way within the security proof of our offline phase. We pack all these properties together in our final definition of a *reusable PCG*. In our publication [93], we present a PCG construction by slightly adapting prior work and prove that it fulfills our new definition.

Psuedorandom correlation functions Besides considering PCGs, we also apply our new insights to pseudorandom correlation functions (PCFs). This primitive was recently introduced by Boyle et al. [45] and extends the idea of PCGs. While PCGs expand the seeds to an entire batch of correlation tuples simultaneously, PCFs allow the parties to evaluate on public input to obtain only a single correlation tuple. Conceptually, this reduced the storage complexity as parties can obtain correlation tuples on the fly instead of storing them after the one-time expansion. We extend the definitional framework of PCFs analogously to the definition of PCGs and present instantiations of our offline phase on both PCGs and PCFs. While conceptually more advanced, PCFs lack concretely efficient implementations. Therefore, we present a thorough evaluation of our offline phase based on PCGs. We refer to [93] for more details on our evaluation and benchmarks.

5.2. Related Work

Anonymous credentials were first introduced by Chaum in 1985 [60]. More than a decade later, in 2001, Camenish and Lysyanskaya proposed the first fully anonymous credential scheme [54]. Subsequently, a numerous theoretical and practical work emerged (e.g., [55, 67, 70, 106, 108], and references within [121]). In 2006, Au et al. [12] introduced the BBS+ signature scheme named after the group signature scheme of Boneh, Boyen, and Shacham [38]. Recently, Tessaro and Zhu [187] showed that the BBS+ signature scheme is also secure when removing the s value.

The distributing of signing algorithms was already considered more than 35 years ago by Desmedt [81]. After early results [76, 147, 179] the use of threshold signature scheme to secure cryptographic wallets of blockchains renewed the interest in threshold signatures [39, 58, 83, 84, 102]. Especially threshold ECDSA is a widely studied topic (e.g., [1, 33, 58, 59, 83, 84], and references within [14]).

The BBS+ signing algorithm was first distributed by Gennaro et al. [103] and Doerner et al. [85]. While [103] focused on threshold issuance of the BBS group signature scheme, their techniques can also be applied to the issuance of BBS+ signature. In contrast, [85] directly proposed a threshold protocol for issuing BBS+ signatures in an anonymous credential scheme. Similar to our approach, Gennaro et al. and Doerner et al. build on the idea of the inversion protocol of Bar-Ilan and Beaver [18]. While we use OLE and VOLE correlations computed from PCGs to realize multiplications of secret shares, both use multiplication protocols. [103] incorporates a two-round multiplication protocol based on a linearly homomorphic encryption scheme, and [85] uses a two-round protocol based on oblivious transfer. Consequently, the response to a signing request requires interaction between the signers in both protocols, which is in contrast to our construction. Since parts of their protocols are message-independent, we can consider these steps as an offline phase. This modeling allows their protocol to split into an interactive offline and non-interactive online phase. While the round complexity of the online phase in this model is identical to our construction, the communication complexity of the offline phase is linear. In contrast, our protocol achieves a non-interactive online and an offline phase with sublinear communication complexity. Our advantages come at the cost of an additional assumption, e.g., a variant of the LPN assumption, and increased computational requirements, both due to the use of PCGs.

Wong et al. [201] presented another threshold BBS+ protocol. Their work focuses on providing strong robustness, which allows parties to obtain a valid signature even if some parties stop responding and other parties join during the computation. In order to achieve strong robustness, their construction builds on a linearly homomorphic encryption scheme with threshold decryption. The resulting construction takes four rounds of communication between the servers. In contrast, our online protocol takes only two rounds and follows the request and response communication pattern. Note that our online protocol requires the client to send the signer set to the servers, who prepare partial signatures corresponding to the signer set. This form of synchronization, called a roll call in [201], is not required by the construction by Wong et al. [201].

Sonnino et al. [182] proposed a threshold anonymous credential scheme, called Coconut, based on the Pointcheval-Sanders (PS) signature scheme [167]. Subsequently, Rial and Piotrowska presented a follow-up of Coconut [172]. In contrast to the BBS+ signing algorithm, the PS signing algorithm consists only of linear operations. Therefore, Coconut achieves a non-interactive online phase without precomputation. On the downside, although Coconut also supports multi-attribute credentials, the public and private key size grows linearly with the number of attributes. In contrast, BBS+ keys always have a constant size. Additionally, BBS+ is more popular than the PS signature scheme. The popularity becomes evident by the standardization efforts highlighted in the motivation of this chapter.

The main building blocks used in our offline phase are pseudorandom correlation generators (PCGs) and functions (PCFs). PCGs were recently introduced by Boyle et al. [42] and further refined and extended by [36, 43, 44, 46, 47, 69, 170]. The research on PCFs was initiated by Boyle et al. [45] and continued by [2, 43, 50, 162]. Further, some work used these primitives to increase the efficiency of general-purpose MPC [82] or tailored MPC protocol [1, 131]. Specifically, Abram et al.[1] presents a threshold ECDSA protocol based on a PCG, and Kondi et al. [131] uses a PCF for realizing two-party deterministic Schnorr signing. [1] specifies a PCG that outputs correlations tailored to the ECDSA signature while we build our offline phase on black-box PCGs/PCFs for OLE and VOLE correlations. Additionally, [1] presents an n-out-of-n protocol instead of supporting a flexible threshold as in our construction. Kondi et al.[131] introduce a new primitive called discrete log PCF to realize a deterministic nonce generation in the Schnorr signing algorithm. The protocol works only in the two-party setting. Both constructions [1, 131] require communication per presignature. Therefore, they either require linear communication in the offline phase or an interactive online phase. This trade-off is in contrast to our scheme, which achieves both properties simultaneously.

6. Conclusion

In this thesis, we contributed to the research on scalability and private smart contract execution for blockchains. Additionally, we applied the decentralization paradigm to the issuing of anonymous credentials. Next, we point out interesting directions for future research in these fields.

Instantiating scalability solution over Bitcoin. In Chapter 3, we presented a new solution, called POSE, for improving the scalability of smart contract-supported blockchains. Our POSE system consists of three types of parties. Operators possess trusted execution environments (TEEs) that are responsible for smart contract execution. Users provide inputs to smart contracts, and a single manager is responsible for managing the system, e.g., keeping track of all registered operators and contracts and maintaining balances for all POSE contracts. We realize this manager via an on-chain smart contract. Realizing the manager via a smart contract has the advantage that no additional trust assumption is made. The downside of this approach is that we can use POSE only for blockchains that support smart contracts.

A potential direction for future research is to adapt our POSE system so that it can also be instantiated over blockchains without smart contract support, e.g., Bitcoin. There are two direct positive effects when realizing POSE over Bitcoin. First, we can open our POSE system to another large user base, as Bitcoin is still the blockchain with the highest market capitalization at the time of writing [137]. Second, by combining Bitcoin with POSE, we can extend the supported capabilities of Bitcoin with the execution of arbitrary smart contracts. This extension would increase the expressiveness of the Bitcoin system.

The main challenge in combining Bitcoin with POSE is to realize the manager differently. One approach is to move the responsibility of managing the system to the TEEs. This approach includes the TEEs checking for on-chain challenges, the number of deposited coins, and more. To perform all these checks, the TEEs must be aware of all blockchain data. Since no on-chain manager exists, we cannot apply the same synchronization mechanism as in the current POSE system. Consequently, the TEEs must either verify the entire blockchain or the system must build on a new synchronization mechanism. On the one hand, it is an interesting research question to explore whether TEEs can verify the entire blockchain state. On the other hand, developing a new synchronization mechanism to verify blockchain data efficiently might be worthwhile.

Privacy via maliciously secure multiparty computation. In Chapter 4, we presented a general-purpose MPC protocol to facilitate private smart contract execution. For all the presented protocols, we considered the covert and publicly verifiable covert (PVC) security setting and integrated mechanism to facilitate automatic verification of proofs of misbehavior within smart contracts. The initial idea of covert security was to lower the security guarantees of the malicious security setting slightly to facilitate more efficient protocols. With more research on general-purpose MPC protocols in the malicious security setting, the efficiency overhead compared to covert protocols diminished. Consequently, using maliciously secure protocols becomes more and more attractive. However, maliciously secure protocols do not provide built-in mechanisms to verify malicious behavior publicly. Therefore, an interesting research question is whether we can leverage our techniques used for covert and PVC protocols and design maliciously secure MPC protocols that allow a smart contract to decide on accused misbehavior. Such protocols would facilitate private smart contract execution using MPC in the malicious security setting.

To this end, two approaches are good starting points. First, Baum et al. [20] presented the idea of publicly auditable MPC. In this setting, the parties produce an audit trail during the protocol computation, and afterward, any third party can verify the correctness of the result. The verification is even possible if all parties were corrupted during the computation. In contrast to correctness, privacy breaks if all parties are corrupted. Therefore, to achieve private smart contract execution, we must assume at least one honest party. Since this assumption is mandatory, we might be able to leverage the assumption for more efficient verification. In the publicly auditable SPZD protocol presented by Baum et al. [20], the verification requires recomputing the entire computation circuit. As this is prohibitively expensive for automatic verification within a smart contract, an interesting question is whether we can exploit the assumption of at least one honest party to improve the efficiency of the verification.

A second potential direction is the line of research on identifiable abort initially studied in [116] (see also [22, 66] and references within). Since a cheating attempt in the malicious security setting is always detected except with negligible probability, such an attempt can also be considered an abort of the cheating party. Therefore, techniques from protocols with identifiable abort could be helpful to identify the cheating party. An open question in this direction is how to transfer the knowledge of an abort in a publicly verifiable way such that a smart contract can verify an accusation.

Improved pseudorandom correlations generation. In Chapter 5, we presented a tailored MPC protocol to distribute the creation of BBS+ signatures, a prominent solution for realizing anonymous credentials. Our main building blocks inside our protocol are pseudorandom correlations generated by pseudorandom correlation generators (PCGs) or functions (PCFs). These primitives were recently introduced by Boyle et al. [42] and [45]. Since then, the primitive was already used to improve the communication complexity of threshold signature schemes (e.g., ECDSA [1] and Schnorr [131]) and general-purpose MPC based on authenticated garbling [82]. These works show the potential of PCGs and PCFs. However, these primitives are still in their infancy and can be improved in multiple directions. First, existing constructions are limited to simple correlations, e.g., OLE, VOLE, OT tuples, and multiplication triples. Building constructions for other correlations would further increase the range of applications or improve existing use cases. For instance, our protocol in Chapter 5 builds on OLE and VOLE correlations, providing additive shares of multiplications. A correlation that provides Shamir-style shares of multiplications would allow us to define the signer set on the client side, increasing the client's flexibility. Second, while PCGs and PCFs provide sublinear communication complexity, their computation and storage complexity are too high for practical use. For instance, the key size for an OLE PCF based on the construction of Boyle et al. [45] is over 2 TB for reasonable parameters. Due to their high potential, we deem further research on PCGs and PCFs as an interesting research direction.

7. Bibliography

[1] D. Abram, A. Nof, C. Orlandi, P. Scholl, and O. Shlomovits. “Low-Bandwidth Threshold ECDSA via Pseudorandom Correlation Generators”. In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022.* 2022, pp. 2554–2572.

[2] D. Abram, P. Scholl, and S. Yakoubov. “Distributed (Correlation) Samplers: How to Remove a Trusted Dealer in One Round”. In: *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part I.* 2022, pp. 790–820.

[3] B. Academy. *What Is Blockchain Network Congestion? — Binance Academy.* `https://academy.binance.com/en/articles/what-is-blockchain-network-congestion`. (Accessed on 06/27/2024). 2023.

[4] H. Amler, L. Eckey, S. Faust, M. Kaiser, P. G. Sandner, and B. Schlosser. “DeFi-ning DeFi: Challenges & Pathway”. In: *3rd Conference on Blockchain Research & Applications for Innovative Networks and Services, BRAINS 2021, Paris, France, September 27-30, 2021.* 2021, pp. 181–184.

[5] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. “Fair Two-Party Computations via Bitcoin Deposits”. In: *Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers.* 2014, pp. 105–121.

[6] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. “Secure Multiparty Computations on Bitcoin”. In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* 2014, pp. 443–458.

[7] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida. “Generalizing the SPDZ Compiler For Other Protocols”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 2018, pp. 880–895.

[8] *Arbitrum — The Future of Ethereum.* `https://arbitrum.io/`. (Accessed on 05/22/2024).

[9] 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.

[10] 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.

[11] 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.

[12] M. H. Au, W. Susilo, Y. Mu, and S. S. M. Chow. “Constant-Size Dynamic k-Times Anonymous Authentication”. In: *IEEE Syst. J.* 2 (2013), pp. 249–261.

[13] 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.

[14] J. Aumasson, A. Hamelink, and O. Shlomovits. “A Survey of ECDSA Threshold Signing”. In: *IACR Cryptol. ePrint Arch.* (2020), p. 1390.

[15] S. Avizheh, P. Haffey, and R. Safavi-Naini. “Privacy-enhanced OptiSwap”. In: *CCSW@CCS ’21: Proceedings of the 2021 on Cloud Computing Security Workshop, Virtual Event, Republic of Korea, 15 November 2021.* 2021, pp. 39–57.

[16] 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.

[17] A. Banerjee and H. Tewari. “Multiverse of HawkNess: A Universally-Composable MPC-Based Hawk Variant”. In: *Cryptogr.* 3 (2022), p. 39.

[18] J. Bar-Ilan and D. Beaver. “Non-Cryptographic Fault-Tolerant Computing in Constant Number of Rounds of Interaction”. In: *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989.* 1989, pp. 201–209.

[19] 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.

[20] 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.

[21] C. Baum, B. David, and R. Dowsley. “Insured MPC: Efficient Secure Computation with Financial Penalties”. In: *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers.* 2020, pp. 404–420.

[22] C. Baum, N. Melissaris, R. Rachuri, and P. Scholl. “Cheater Identification on a Budget: MPC with Identifiable Abort from Pairwise MACs”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1548.

[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] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* 2014, pp. 459–474.

[26] I. Bentov and R. Kumaresan. “How to Use Bitcoin to Design Fair Protocols”. In: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II.* 2014, pp. 421–439.

[27] K. Bergman and S. Rajput. “Revealing and Concealing Bitcoin Identities: A Survey of Techniques”. In: *BSCI ’21: Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, Virtual Event, Hong Kong, June 7, 2021.* 2021, pp. 13–24.

[28] G. Bernstein and M. Sporny. *Data Integrity BBS Cryptosuites v1.0.* `https://w3c.github.io/vc-di-bbs/`. (Accessed on 05/13/2024). 2024.

[29] N. Bitansky, S. Goldwasser, A. Jain, O. Paneth, V. Vaikuntanathan, and B. Waters. “Time-Lock Puzzles from Randomized Encodings”. In: *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016.* 2016, pp. 345–356.

[30] *Bitcoin — Active Addresses Count Chart — Messari.* `https://messari.io/project/bitcoin/charts/addresses/chart/act-addr-cnt`. (Accessed on 04/17/2024).

[31] *Bitcoin Cash.* `https://bitcoincash.org/`. (Accessed on 05/21/2024).

[32] *Bitcoin, Ethereum, Dogecoin, Litecoin stats.* `https://bitinfocharts.com/`. (Accessed on 04/09/2024).

[33] C. Blokh, N. Makriyannis, and U. Peled. "Efficient Asymmetric Threshold ECDSA for MPC-based Cold Storage". In: *IACR Cryptol. ePrint Arch.* (2022), p. 1296.

[34] *BNB Chain.* `https://www.bnbchain.org/en`. (Accessed on 05/22/2024).

[35] D. van Bokkem, R. Hageman, G. Koning, L. Nguyen, and N. Zarin. "Self-Sovereign Identity Solutions: The Necessity of Blockchain Technology". In: *CoRR* (2019). arXiv: `1904.12816`.

[36] M. Bombar, G. Couteau, A. Couvreur, and C. Ducros. "Correlated Pseudorandomness from the Hardness of Quasi-Abelian Decoding". In: *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV.* 2023, pp. 567–601.

[37] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. "Verifiable Delay Functions". In: *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I.* 2018, pp. 757–788.

[38] D. Boneh, X. Boyen, and H. Shacham. "Short Group Signatures". In: *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings.* 2004, pp. 41–55.

[39] D. Boneh, M. Drijvers, and G. Neven. "Compact Multi-signatures for Smaller Blockchains". In: *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II.* 2018, pp. 435–464.

[40] 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.

[41] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. "ZEXE: Enabling Decentralized Private Computation". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* 2020, pp. 947–964.

[42] 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.

[43] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, N. Resch, and P. Scholl. “Correlated Pseudorandomness from Expand-Accumulate Codes”. In: *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part II*. 2022, pp. 603–633.

[44] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, N. Resch, and P. Scholl. “Oblivious Transfer with Constant Computational Overhead”. In: *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part I*. 2023, pp. 271–302.

[45] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. “Correlated Pseudorandom Functions from Variable-Density LPN”. In: *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. 2020, pp. 1069–1080.

[46] 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.

[47] 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.

[48] E. Brickell and J. Li. “A Pairing-Based DAA Scheme Further Reducing TPM Resources”. In: *Trust and Trustworthy Computing, Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*. 2010, pp. 181–195.

[49] E. Brickell and J. Li. “Enhanced privacy ID from bilinear pairing for hardware authentication and attestation”. In: *Int. J. Inf. Priv. Secur. Integr.* 1 (2011), pp. 3–33.

[50] D. Bui, G. Couteau, P. Meyer, A. Passelègue, and M. Riahinia. “Fast Public-Key Silent OT and More from Constrained Naor-Reingold”. In: *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part VI.* 2024, pp. 88–118.

[51] 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.

[52] 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.

[53] J. Camenisch, M. Drijvers, and A. Lehmann. “Anonymous Attestation Using the Strong Diffie Hellman Assumption Revisited”. In: *Trust and Trustworthy Computing - 9th International Conference, TRUST 2016, Vienna, Austria, August 29-30, 2016, Proceedings.* 2016, pp. 1–20.

[54] J. Camenisch and A. Lysyanskaya. “An Efficient System for Non-transferable Anonymous Credentials with Optional Anonymity Revocation”. In: *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding.* 2001, pp. 93–118.

[55] J. Camenisch and A. Lysyanskaya. “Signature Schemes and Anonymous Credentials from Bilinear Maps”. In: *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings.* 2004, pp. 56–72.

[56] 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.

[57] *Cardano.* `https://cardano.org/`. (Accessed on 04/09/2024).

[58] G. Castagnos, D. Catalano, F. Laguillaumie, F. Savasta, and I. Tucker. “Bandwidth-Efficient Threshold EC-DSA”. In: *Public-Key Cryptography - PKC 2020 - 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4-7, 2020, Proceedings, Part II.* 2020, pp. 266–296.

[59] G. Castagnos, D. Catalano, F. Laguillaumie, F. Savasta, and I. Tucker. “Bandwidth-efficient threshold EC-DSA revisited: Online/offline extensions, identifiable aborts proactive and adaptive security”. In: *Theor. Comput. Sci.* (2023), pp. 78–104.

[60] D. Chaum. “Security Without Identification: Transaction Systems to Make Big Brother Obsolete”. In: *Commun. ACM* 10 (1985), pp. 1030–1044.

[61] L. Chen. “A DAA Scheme Requiring Less TPM Resources”. In: *Information Security and Cryptology - 5th International Conference, Inscrypt 2009, Beijing, China, December 12-15, 2009. Revised Selected Papers.* 2009, pp. 350–365.

[62] Y. Chen, Y. Lu, L. Bulysheva, and M. Y. Kataev. “Applications of blockchain in industry 4.0: A review”. In: *Information Systems Frontiers* (2022), pp. 1–15.

[63] 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.

[64] A. R. Choudhuri, V. Goyal, and A. Jain. “The Round Complexity of Secure Computation Against Covert Adversaries”. In: *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings.* 2020, pp. 600–620.

[65] T. Close. “Nitro Protocol”. In: *IACR Cryptol. ePrint Arch.* (2019), p. 219.

[66] R. Cohen, J. Doerner, Y. Kondi, and A. Shelat. “Secure Multiparty Computation with Identifiable Abort from Vindicating Release”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1136.

[67] A. Connolly, J. Deschamps, P. Lafourcade, and O. Perez-Kempner. “Protego: Efficient, Revocable and Auditable Anonymous Credentials with Applications to Hyperledger Fabric”. In: *Progress in Cryptology - INDOCRYPT 2022 - 23rd International Conference on Cryptology in India, Kolkata, India, December 11-14, 2022, Proceedings.* 2022, pp. 249–271.

[68] *Counterfactual: Generalized State Channels.* `https://ethresear.ch/t/counterfactual-generalized-state-channels/2223`. (Accessed on 05/22/2024). 2018.

[69] 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.

[70] E. C. Crites and A. Lysyanskaya. “Delegatable Anonymous Credentials from Mercurial Signatures”. In: *Topics in Cryptology - CT-RSA 2019 - The Cryptographers’ Track at the RSA Conference 2019, San Francisco, CA, USA, March 4-8, 2019, Proceedings.* 2019, pp. 535–555.

[71] *CryptoKitties.* `https://www.cryptokitties.co/`. (Accessed on 04/09/2024).

[72] W. Dai. “PESCA: A Privacy-Enhancing Smart-Contract Architecture”. In: *IACR Cryptol. ePrint Arch.* (2022), p. 1119.

[73] 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.

[74] 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.

[75] 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.

[76] I. Damgård and M. Koprowski. “Practical Threshold RSA Signatures without a Trusted Dealer”. In: *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding.* 2001, pp. 152–165.

[77] 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.

[78] 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.

[79] 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.

[80] S. Das, V. J. Ribeiro, and A. Anand. “YODA: Enabling computationally intensive contracts on blockchains with Byzantine and Selfish nodes”. In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.* 2019.

[81] Y. Desmedt. “Society and Group Oriented Cryptography: A New Concept”. 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. 120–127.

[82] 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.

[83] J. Doerner, Y. Kondi, E. Lee, and A. Shelat. “Secure Two-party Threshold ECDSA from ECDSA Assumptions”. In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA.* 2018, pp. 980–997.

[84] J. Doerner, Y. Kondi, E. Lee, and A. Shelat. “Threshold ECDSA from ECDSA Assumptions: The Multiparty Case”. In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019.* 2019, pp. 1051–1066.

[85] J. Doerner, Y. Kondi, E. Lee, A. Shelat, and L. Tyner. “Threshold BBS+ Signatures for Distributed Anonymous Credential Issuance”. In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023.* 2023, pp. 773–789.

[86] L. Duan, Y. Jiang, Y. Li, J. Müller-Quade, and A. Rupp. “Security Against Honorific Adversaries: Efficient MPC with Server-aided Public Verifiability”. In: *IACR Cryptol. ePrint Arch.* (2022), p. 606.

[87] C. Dwork and M. Naor. “Pricing via Processing or Combatting Junk Mail”. In: *Advances in Cryptology - CRYPTO ’92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings.* 1992, pp. 139–147.

[88] 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.

[89] 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.

[90] 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.

[91] L. Eckey, S. Faust, and B. Schlosser. “OptiSwap: Fast Optimistic Fair Exchange”. In: *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020.* 2020, pp. 543–557.

[92] *EOSIO Blockchain Software & Services.* `https://eos.io/`. (Accessed on 04/09/2024).

[93] S. Faust, C. Hazay, D. Kretzler, L. Rometsch, and B. Schlosser. *Non-Interactive Threshold BBS+ From Pseudorandom Correlations.* Cryptology ePrint Archive, Paper 2023/1076. `https://eprint.iacr.org/2023/1076`. 2023. **Part of this thesis**.

[94] 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**.

[95] 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**.

[96] 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**.

[97] 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.

[98] D. Feng and K. Yang. “Concretely efficient secure multi-party computation protocols: survey and more”. In: *Security and Safety* (2022), p. 2021001.

[99] 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**.

[100] 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.

[101] S. D. Galbraith, K. G. Paterson, and N. P. Smart. "Pairings for cryptographers". In: *Discret. Appl. Math.* 16 (2008), pp. 3113–3121.

[102] R. Gennaro and S. Goldfeder. "Fast Multiparty Threshold ECDSA with Fast Trustless Setup". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 2018, pp. 1179–1194.

[103] R. Gennaro, S. Goldfeder, and B. Ithurburn. *Fully distributed group signatures.* 2019.

[104] E. Georgiadis. "How many transactions per second can bitcoin really handle ? Theoretically". In: *IACR Cryptol. ePrint Arch.* (2019), p. 416.

[105] N. Gilboa. "Two Party RSA Key Generation". In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings.* 1999, pp. 116–129.

[106] *GitHub - IBM/idemix: implementation of an anonymous identity stack for blockchain systems.* `https://github.com/IBM/idemix`. (Accessed on 05/23/2024).

[107] V. Goyal, P. Mohassel, and A. D. Smith. "Efficient Two Party and Multi Party Computation Against Covert Adversaries". In: *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings.* 2008, pp. 289–306.

[108] T. Group et al. "Trusted platform module library part 1: Architecture". In: *Trusted Computing Group, Tech. Rep* (2019).

[109] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. "SoK: Layer-Two Blockchain Protocols". In: *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers.* 2020, pp. 201–226.

[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.

[112] 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.

[113] Y. Huang, J. Katz, and D. Evans. "Quid-Pro-Quo-tocols: Strengthening Semi-honest Protocols with Dual Execution". In: *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA.* 2012, pp. 272–284.

[114] *Hyperledger Indy.* `https://www.hyperledger.org/projects/hyperledger-indy`. (Accessed on 04/25/2024).

[115] Y. Ishai, E. Kushilevitz, R. Ostrovsky, M. Prabhakaran, and A. Sahai. "Efficient Non-interactive Secure Computation". In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings.* 2011, pp. 406–425.

[116] Y. Ishai, R. Ostrovsky, and H. Seyalioglu. "Identifying Cheaters without an Honest Majority". In: *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings.* 2012, pp. 21–38.

[117] 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.

[118] T. P. Jakobsen, J. B. Nielsen, and C. Orlandi. "A Framework for Outsourcing of Secure Computation". In: *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014.* 2014, pp. 81–92.

[119] M. Jakobsson and A. Juels. "Proofs of Work and Bread Pudding Protocols". In: *Secure Information Networks: Communications and Multimedia Security, IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS '99), September 20-21, 1999, Leuven, Belgium.* 1999, pp. 258–272.

[120] M. Jourenko, K. Kurazumi, M. Larangeira, and K. Tanaka. "SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies". In: *IACR Cryptol. ePrint Arch.* (2019), p. 352.

[121] S. A. Kakvi, K. M. Martin, C. Putman, and E. A. Quaglia. "SoK: Anonymous Credentials". In: *Security Standardisation Research - 8th International Conference, SSR 2023, Lyon, France, April 22-23, 2023, Proceedings.* 2023, pp. 129–151.

[122] 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.

[123] S. Kamara, P. Mohassel, and B. Riva. "Salus: a system for server-aided secure function evaluation". In: *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012.* 2012, pp. 797–808.

[124] T. Kerber, A. Kiayias, and M. Kohlweiss. "KACHINA - Foundations of Private Smart Contracts". In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021.* 2021, pp. 1–16.

[125] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais. *Commit-Chains: Secure, Scalable Off-Chain Payments.* Cryptology ePrint Archive, Paper 2018/642. `https://eprint.iacr.org/2018/642`. 2018.

[126] M. C. K. Khalilov and A. Levi. "A Survey on Anonymity and Privacy in Bitcoin-Like Digital Cash Systems". In: *IEEE Commun. Surv. Tutorials* 3 (2018), pp. 2543–2585.

[127] S. King and S. Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake.* `https://www.peercoin.net/read/papers/peercoin-paper.pdf`. (Accessed on 05/27/2024). 2012.

[128] P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019.* 2019, pp. 1–19.

[129] 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.

[130] 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.

[131] Y. Kondi, C. Orlandi, and L. Roy. "Two-Round Stateless Deterministic Two-Party Schnorr Signatures from Pseudorandom Correlation Functions". In: *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part I.* 2023, pp. 646–677.

[132] 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.

[133] R. Kumaresan and I. Bentov. “Amortizing Secure Computation with Penalties”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, pp. 418–429.

[134] R. Kumaresan, T. Moran, and I. Bentov. “How to Use Bitcoin to Play Decentralized Poker”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 2015, pp. 195–206.

[135] R. Kumaresan, V. Vaikuntanathan, and P. N. Vasudevan. “Improvements to Secure Computation with Penalties”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, pp. 406–417.

[136] 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.

[137] *Largest Blockchains in Crypto Ranked by TVL — CoinMarketCap*. `https://coinmarketcap.com/chain-ranking/`. (Accessed on 04/09/2024).

[138] 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.

[139] 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.

[140] 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.

[141] Y. Lindell and B. Pinkas. “An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries”. In: *Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings*. 2007, pp. 52–78.

[142] M. Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018, pp. 973–990.

[143] 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.

[144] 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.

[145] M. Lohr, B. Schlosser, J. Jürjens, and S. Staab. “Cost Fairness for Blockchain-Based Two-Party Exchange Protocols”. In: *IEEE International Conference on Blockchain, Blockchain 2020, Rhodes, Greece, November 2-6, 2020*. 2020, pp. 428–435.

[146] T. Looker, V. Kalos, A. Whitehead, and M. Lodder. *The BBS Signature Scheme*. Internet-Draft draft-irtf-cfrg-bbs-signatures-05. Work in Progress. Internet Engineering Task Force, 2023. 115 pp.

[147] P. D. MacKenzie and M. K. Reiter. “Two-Party Generation of DSA Signatures”. In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*. 2001, pp. 137–154.

[148] M. Mahmoody, T. Moran, and S. P. Vadhan. “Time-Lock Puzzles in the Random Oracle Model”. In: *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*. 2011, pp. 39–50.

[149] G. Malavolta and S. A. K. Thyagarajan. “Homomorphic Time-Lock Puzzles and Applications”. In: *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I*. 2019, pp. 620–649.

[150] MATTR. *GitHub - mattrglobal/bbs-signatures: An implementation of BBS+ signatures for node and browser environments*. `https://github.com/mattrglobal/bbs-signatures`. (Accessed on 05/13/2024).

[151] P. McCorry, S. Bakshi, I. Bentov, S. Meiklejohn, and A. Miller. “Pisa: Arbitration Outsourcing for State Channels”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019.* 2019, pp. 16–30.

[152] P. McCorry, C. Buckland, S. Bakshi, K. Wüst, and A. Miller. “You Sank My Battleship! A Case Study to Evaluate State Channels as a Scaling Solution for Cryptocurrencies”. In: *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers.* 2019, pp. 35–49.

[153] Microsoft. *microsoft/bbs-node-reference: TypeScript/node reference implementation of BBS signature.* `https://github.com/microsoft/bbs-node-reference`. (Accessed on 05/13/2024).

[154] 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.

[155] P. Mohassel and M. K. Franklin. “Efficiency Tradeoffs for Malicious Two-Party Computation”. In: *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings.* 2006, pp. 458–473.

[156] 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.

[157] A. Mühle, A. Grüner, T. Gayvoronskaya, and C. Meinel. “A survey on essential components of a self-sovereign identity”. In: *Comput. Sci. Rev.* (2018), pp. 80–86.

[158] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008.

[159] 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.

[160] *Optimism.* `https://optimism.io/`. (Accessed on 05/21/2024).

[161] *Optimistic Rollups — ethereum.org.* `https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/`. (Accessed on 05/21/2024).

[162] C. Orlandi, P. Scholl, and S. Yakoubov. “The Rise of Paillier: Homomorphic Secret Sharing and Public-Key Silent OT”. 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 I*. 2021, pp. 678–708.

[163] R. Pass, E. Shi, and F. Tramèr. “Formal Abstractions for Attested Execution Secure Processors”. In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*. 2017, pp. 260–289.

[164] 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.

[165] 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.

[166] *Plasma chains — ethereum.org*. `https://ethereum.org/en/developers/docs/scaling/plasma/`. (Accessed on 05/23/2024).

[167] D. Pointcheval and O. Sanders. “Short Randomizable Signatures”. In: *Topics in Cryptology - CT-RSA 2016 - The Cryptographers’ Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*. 2016, pp. 111–126.

[168] J. Poon and V. Buterin. “Plasma: Scalable Autonomous Smart Contracts”. In: (2017).

[169] H. Qi, M. Xu, D. Yu, and X. Cheng. “SoK: Privacy-Preserving Smart Contract”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1226.

[170] S. Raghuraman, P. Rindal, and T. Tanguy. “Expand-Convolute Codes for Pseudorandom Correlation Generators from LPN”. In: *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV*. 2023, pp. 602–632.

[171] Q. Ren, Y. Wu, H. Liu, Y. Li, A. Victor, H. Lei, L. Wang, and B. Chen. “Cloak: Transitioning States on Legacy Blockchains Using Secure and Publicly Verifiable Off-Chain Multi-Party Computation”. In: *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. 2022, pp. 117–131.

[172] A. Rial and A. M. Piotrowska. “Security Analysis of Coconut, an Attribute-Based Credential Scheme with Threshold Issuance”. In: *IACR Cryptol. ePrint Arch.* (2022), p. 11.

[173] R. L. Rivest, A. Shamir, and D. A. Wagner. *Time-lock puzzles and timed-release crypto.* Tech. rep. Massachusetts Institute of Technology. Laboratory for Computer Science, 1996.

[174] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer. “Scalable and Probabilistic Leaderless BFT Consensus through Metastability”. In: *CoRR* (2019). arXiv: `1906.08936`.

[175] N. van Saberhagen. *CryptoNote v2.0.* `https://bytecoin.org/old/whitepaper.pdf`. (Accessed on 04/09/2024). 2013.

[176] 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.

[177] *Self-Sovereign Identity: The Ultimate Guide 2024.* `https://www.dock.io/post/self-sovereign-identity`. (Accessed on 04/10/2024).

[178] A. Shamir. “How to Share a Secret”. In: *Commun. ACM* 11 (1979), pp. 612–613.

[179] V. Shoup. “Practical Threshold Signatures”. In: *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding.* 2000, pp. 207–220.

[180] *Solana.* `https://solana.com/de`. (Accessed on 05/22/2024).

[181] R. Solomon, R. Weber, and G. Almashaqbeh. “smartFHE: Privacy-Preserving Smart Contracts from Fully Homomorphic Encryption”. In: *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023.* 2023, pp. 309–331.

[182] A. Sonnino, M. Al-Bassam, S. Bano, S. Meiklejohn, and G. Danezis. “Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers”. In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.* 2019.

[183] *Starknet.* `https://www.starknet.io/`. (Accessed on 05/22/2024).

[184] 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.

[185] 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.

[186] S. Steffen, B. Bichsel, and M. T. Vechev. “Zapper: Smart Contracts with Data and Identity Privacy”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022.* 2022, pp. 2735–2749.

[187] S. Tessaro and C. Zhu. “Revisiting BBS Signatures”. In: *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part V.* 2023, pp. 691–721.

[188] J. Teutsch and C. Reitwießner. “A scalable verification solution for blockchains”. In: *CoRR* (2019). arXiv: `1908.04756`.

[189] L. T. Thibault, T. Sarry, and A. S. Hafid. “Blockchain Scaling Using Rollups: A Comprehensive Survey”. In: *IEEE Access* (2022), pp. 93039–93054.

[190] *Transaction size calculator — Bitcoin Optech.* `https://bitcoinops.org/en/tools/calc-size/`. (Accessed on 04/09/2024).

[191] Trinsic. *Credential API - Documentation.* `https://docs.trinsic.id/reference/services/credential-service/`. (Accessed on 05/13/2024).

[192] *VISA Fact Sheet.* `https://www.visa.co.uk/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf`. (Accessed on 04/09/2024).

[193] G. Wang, Z. J. Shi, M. Nixon, and S. Han. “SoK: Sharding on Blockchain”. In: *Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019.* 2019, pp. 41–61.

[194] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F. Wang. “An Overview of Smart Contract: Architecture, Applications, and Future Trends”. In: *2018 IEEE Intelligent Vehicles Symposium, IV 2018, Changshu, Suzhou, China, June 26-30, 2018.* 2018, pp. 108–113.

[195] X. Wang, A. J. Malozemoff, and J. Katz. “Faster Secure Two-Party Computation in the Single-Execution Setting”. In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III.* 2017, pp. 399–424.

[196] 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.

[197] X. Wang, S. Ranellucci, and J. Katz. “Global-Scale Secure Multiparty 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. 39–56.

[198] N. Webb. “A fork in the blockchain: income tax and the bitcoin/bitcoin cash hard fork”. In: *North Carolina Journal of Law & Technology* 4 (2018), p. 283.

[199] 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.

[200] *What Is The Bitcoin Block Size Limit? - Bitcoin Magazine - Bitcoin News, Articles and Expert Insights.* `https://bitcoinmagazine.com/guides/what-is-the-bitcoin-block-size-limit`. (Accessed on 04/09/2024).

[201] H. W. H. Wong, J. P. K. Ma, and S. S. M. Chow. “Secure Multiparty Computation of Threshold Signatures Made More Efficient”. In: *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024.* 2024.

[202] G. Wood. *Polkadot: Vision for a heterogeneous multi-chain framework.* 2016.

[203] G. Wood et al. “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper* (2014).

[204] K. Wüst, L. Diana, K. Kostiainen, G. Karame, S. Matetic, and S. Capkun. “Bitcontracts: Adding Expressive Smart Contracts to Legacy Cryptocurrencies”. In: *IACR Cryptol. ePrint Arch.* (2019), p. 857.

[205] K. Wüst, S. Matetic, S. Egli, K. Kostiainen, 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.

[206] A. L. Xiong, B. Chen, Z. Zhang, B. Bünz, B. Fisch, F. Krell, and P. Camacho. “VERI-ZEXE: Decentralized Private Computation with Universal Setup”. In: *IACR Cryptol. ePrint Arch.* (2022), p. 802.

[207] 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.

[208] R. Yuan, Y. Xia, H. Chen, B. Zang, and J. Xie. "ShadowEth: Private Smart Contract on Public Blockchain". In: *J. Comput. Sci. Technol.* 3 (2018), pp. 542–556.

[209] *Zcash — Active Addresses Count Chart — Messari.* `https://messari.io/project/zcash/charts/addresses/chart/act-addr-cnt`. (Accessed on 04/17/2024).

[210] B. Zeng, C. Tartary, P. Xu, J. Jing, and X. Tang. "A Practical Framework for t-Out-of-n Oblivious Transfer With Security Against Covert Adversaries". In: *IEEE Trans. Inf. Forensics Secur.* 2 (2012), pp. 465–479.

[211] *Zero-knowledge rollups — ethereum.org.* `https://ethereum.org/en/developers/docs/scaling/zk-rollups/`. (Accessed on 05/22/2024).

[212] F. Zhang, W. He, R. Cheng, J. Kos, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song. "The Ekiden Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts". In: *IEEE Secur. Priv.* 3 (2020), pp. 17–27.

[213] 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.

[214] *zkSync.* `https://zksync.io/`. (Accessed on 05/22/2024).

[215] G. Zyskind, O. Nathan, and A. Pentland. "Enigma: Decentralized Computation Platform with Guaranteed Privacy". In: *CoRR* (2015). arXiv: `1506.03471`.

List of Figures

List of Abbreviations

BRC	Bitcoin Request for Comment
DeFi	Decentralized Finance
ECDSA	Elliptic Curve Digital Signature Algorithm
FBC	Financially Backed Covert
FHE	Fully Homomorphic Encryption
LPN	Learning Parity with Noise
MAC	Message Authentication Code
MPC	Multiparty Computation
OLE	Oblivious Linear Evaluation
OT	Oblivious Transfer
PCG	Pseudorandom Correlation Generator
PCF	Pseudorandom Correlation Function
PFE	Private Function Evaluation
PoS	Proof-of-Stake
POSE	Practical Off-chain Smart Contract Execution
PoW	Proof-of-Work
PRG	Pseudorandom Generator
PVC	Publicly Verifiable Covert
q**-SDH**	q-strong Diffie-Hellman assumption
rPCF	reusable Pseudorandom Correlation Function
SGX	Software Guard Extension
SSI	Self-Sovereign Identities
TEE	Trusted Execution Environment
TLP	Time-Lock Puzzle
VDF	Verifiable Delay Function
VOLE	Vector Oblivious Linear Evaluation
VTLP	Verifiable Time-Lock Puzzle
ZKP	Zero-Knowledege Proof

A. POSE: Practical Off-chain Smart Contract Execution

In this chapter, we present the following publication with minor changes.

[99] 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[1], Patrick Jauernig[1], David Koisser[1], David Kretzler[2], Benjamin Schlosser[2], Sebastian Faust[2], and Ahmaz-Reza Sadeghi[1]

[1] Technical University of Darmstadt, Germany
{first.last}@trust.tu-darmstadt.de
[2] Technical University of Darmstadt, Germany
{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.

1 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: $\boxed{\text{i}}$ participating parties need to lock large amounts of collateral; $\boxed{\text{ii}}$ costly blockchain interactions are required at every step of the process or at regular intervals; and finally $\boxed{\text{iii}}$ 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 $\boxed{\text{iv}}$ 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 2 for an overview on related work and to Section 10 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. $\boxed{\text{i}}$). 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. $\boxed{\text{iii}}$). 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. $\boxed{\text{ii}}$). 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. $\boxed{\text{iv}}$). 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 3.

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. $\boxed{\text{i}}$). The state remains confidential, which allows *POSE* to have private state (cf. $\boxed{\text{iv}}$). 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. $\boxed{\text{iii}}$). Above all, *POSE* executes smart contracts quickly and efficiently without any blockchain interactions in the optimistic case (cf. $\boxed{\text{ii}}$).

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.

2 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 7.2.
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.

3 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 practical[3]. 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.

3.1 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 5.5. 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*

[3] We design *POSE* without depending on any specific TEE implementation. In Section 7.2, we discuss the implications of using ARM TrustZone to realize our scheme.

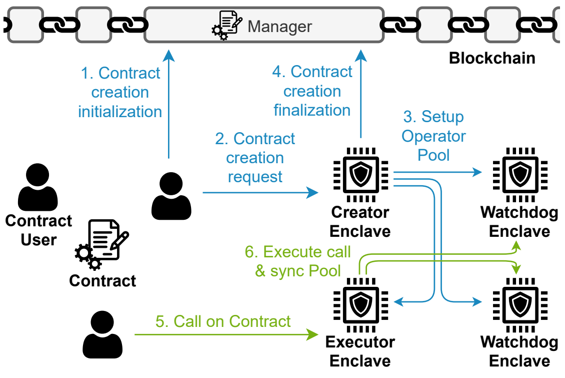


Fig. 1. Exemplary overview how *POSE* contracts are created (in blue) and executed (in green).

enclave, which is responsible for executing the *POSE* contract, as well as the *watchdogs*, ensuring availability. We elaborate on this in-depth in Section 5.5. In Step 5, another user can 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 5.5 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 5.5). 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.

3.2 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 2). 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 7.1 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 5.4.

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.

4 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 $\mathit{Verify}(m')$ 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 $\mathit{TEE.install}(\mathit{prog})$ 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 $\mathit{TEE.resume}(\mathit{eid}, \mathit{inp})$ 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 $(\mathit{eid}, \mathit{prog})$. ρ can be verified by using method $\mathit{VerifyQuote}(\rho)$. We consider only one instance $\mathcal{E}$ running the *POSE* program per TEE. Therefore, we simplify the notation and write $\mathcal{E}(\mathit{inp})$ for $\mathit{TEE.resume}(\mathit{eid}, \mathit{inp})$.

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 5.5, enclaves consider only transactions included in final blocks. Finally, we define that any smart contact deployed to the blockchain is able to access the current timestamp using the method $\mathsf{BC}.\mathit{now}$ and the hash of the most recent 265 blocks [7] using the method $\mathsf{BC}.\mathit{bh}(i)$ where i is the number of the accessed block. These features are available on Ethereum.

5 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.

5.1 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 ($\mathit{creator}$), a hash of the program code ($\mathit{codeHash}$), the set of enclaves forming the execution pool (pool), a total amount of locked coins ($\mathit{balance}$), 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.

5.2 *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 5.5: 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 5.4). 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 5.3 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 2), 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. (cf. Section 5.3). This prevents replay attacks against both, executive and watchdog enclaves.

5.3 *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 5.4, 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 $\mathit{flag} = \mathit{all}$, it returns the whole smart contract state. Otherwise, if called with $\mathit{flag} = \mathit{pub}$, 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

Program 1: *POSE* Program (execution) executed by enclave T

```
Upon invocation with input blockchain data BC, store BC.
Upon receiving $m := (\texttt{execute}, id, r, move; U)$, do:

1. If $M^{id}.pool[0] \neq T$ or $\mathcal{T}^{id}_{wait} \neq \emptyset$, return $(\texttt{bad})$.
2. Execute $C_{id}.nextState(U, \mathsf{BC}, move, H(m))$.
3. Store $\mathcal{T}^{id}_{wait} = M^{id}.pool$ and $h^{id} = H(m)$, set $c = Enc(C_{id}.getState(all); key^{id})$ and return
   $(\texttt{update}, id, c, h^{id}; T)$.

Upon receiving $m := (\texttt{update}, id, c, h; T')$, do:

1. If $T' \neq M^{id}.pool[0]$ or $T \notin M^{id}.pool$, return $(\texttt{bad})$.
2. Define $state = Dec(c; key^{id})$ and call $C_{id}.update(state, h)$.
3. Return $(\texttt{confirm}, id, h; T)$.

Upon receiving $\{m_i := (\texttt{confirm}, id, h_i; T_i)\}_i$, do:

1. If $M^{id}.pool[0] \neq T$ or $\mathcal{T}^{id}_{wait} = \emptyset$, return $(\texttt{bad})$.
2. Set $\mathcal{T}^{id}_{wait} = \mathcal{T}^{id}_{wait} \cap M^{id}.pool$.
3. For each $m_i$ do:
   – If $h_i \neq h^{id}$ or $T_i \notin \mathcal{T}^{id}_{wait}$, skip $m_i$.
   – Otherwise remove $T_i$ from $\mathcal{T}^{id}_{wait}$.
4. If $\mathcal{T}^{id}_{wait} \neq \{T\}$, return $(\texttt{bad})$. Otherwise, set $\mathcal{T}^{id}_{wait} = \emptyset$, $state := C_{id}.getState(pub)$ and return
   $(\texttt{ok}, id, state, h^{id}; T)$.
```

Program 2: Interface of a contract C executed within a *POSE* enclave

```
Function: $nextState(U, \mathsf{BC}, move, h)$
Function: $update(new, h)$
Function: $getState(flag)$
```

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 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 [4].

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.

[4] In 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.

5.4 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 2).

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 5.5) 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* [5]. 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

[5] We 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.

updates $\mathit{incrTxHash}$, such that $\mathit{incrTxHash}_{i+1}$ is defined as

$$H(\mathit{incrTxHash}_i \mid\mid tx.\mathit{data} \mid\mid tx.\mathit{sender} \mid\mid tx.\mathit{value})$$

where $tx.\mathit{data}$ is the raw data of tx, $tx.\mathit{sender}$ denotes the creator of tx, and $tx.\mathit{value}$ 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 $\mathit{incrTxHash}$. In order to verify that the on-chain $\mathit{incrTxHash}$ is indeed part of the main chain, operators additionally provide a Merkle proof showing that $\mathit{incrTxHash}$ 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.

5.5 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 5.2. 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.

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 5.2. 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.[6]

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 5.4) 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 $m := (\mathtt{register}, \mathcal{E}_O, \rho_O, \rho_O^{\mathsf{BC}}; O)$ 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 τ_{slack}^{on} blocks. If the check holds and the signature of the operator is valid, i.e., $\mathit{Verify}(m) = \mathtt{ok}$, M adds $\mathcal{E}_O$ (identified by its public key pk_O) to the set of registered enclaves, i.e., $M.\mathit{registered} := M.\mathit{registered} \cup \{\mathcal{E}_O\}$. 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.

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].

[6] An 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].

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 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 5.5).

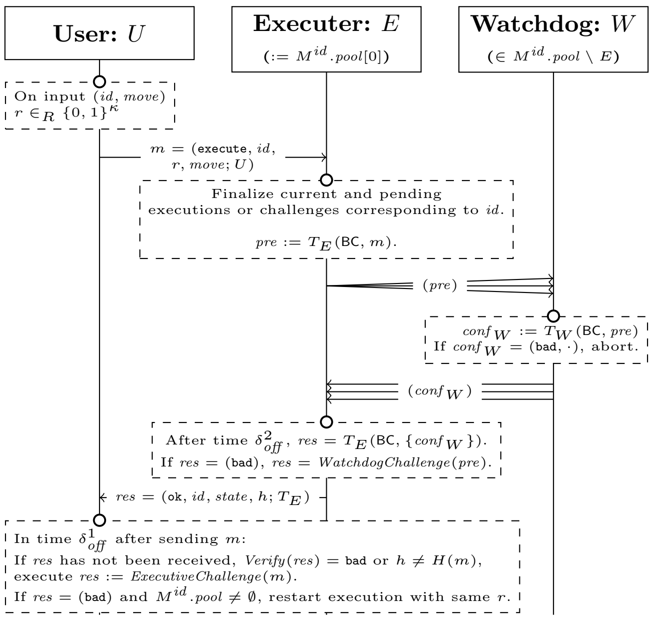

User: U
Executer: E
$(:= M^{id}.pool[0])$
Watchdog: W
$(\in M^{id}.pool \setminus E)$
On input $(id, move)$
$r \in_R \{0,1\}^\kappa$
$m = (\mathtt{execute}, id, r, move; U)$
Finalize current and pending executions or challenges corresponding to id.
$pre := T_E(\mathsf{BC}, m)$.
(pre)
$conf_W := T_W(\mathsf{BC}, pre)$
If $conf_W = (\mathtt{bad}, \cdot)$, abort.
$(conf_W)$
After time δ^2_{off}, $res = T_E(\mathsf{BC}, \{conf_W\})$.
If $res = (\mathtt{bad})$, $res = WatchdogChallenge(pre)$.
$res = (\mathtt{ok}, id, state, h; T_E)$
In time δ^1_{off} after sending m:
If res has not been received, $Verify(res) = \mathtt{bad}$ or $h \neq H(m)$, execute $res := ExecutiveChallenge(m)$.
If $res = (\mathtt{bad})$ and $M^{id}.pool \neq \emptyset$, restart execution with same r.

Fig. 2. Detailed execution protocol.

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., $\mathit{update} = \mathit{update}'$, or a concurrent one, i.e., $\mathit{update} \neq \mathit{update}'$. 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 5.3). 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.

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 ($\delta^{\dagger}_{\star}$), where $\dagger \in \{0, 1\}$ and $\star \in \{\mathit{off}, \mathit{on}\}$, which define the notion of *timely* in Appendix D. In particular, we describe the relation between δ^1_* and δ^2_*. The challenge-response procedure is executed in all of the following cases.

(a) The creator enclave has not responded to the user within time δ^1_{off} during the contract creation protocol.
(b) At least one pool enclave has not responded to the creator enclave within time δ^2_{off} during the contract creation protocol.
(c) The executor enclave has not responded to the user within time δ^1_{off} during the contract execution protocol.
(d) At least one watchdog enclave has not responded to the executor enclave within time δ^2_{off} 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^1_{off}, 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^1_{on} 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 5.5. 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

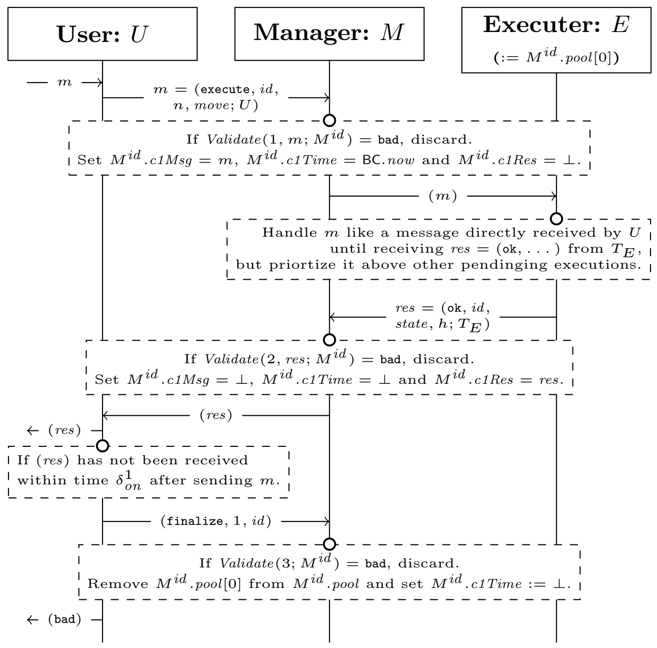

User: U
Manager: M
Executer: E
$(:= M^{id}.pool[0])$
m
$m = (\texttt{execute}, id, n, move; U)$
If $Validate(1, m; M^{id}) = \texttt{bad}$, discard.
Set $M^{id}.c1Msg = m$, $M^{id}.c1Time = \textsf{BC}.now$ and $M^{id}.c1Res = \perp$.
(m)
Handle m like a message directly received by U until receiving $res = (\texttt{ok}, \ldots)$ from T_E, but priortize it above other pendinging executions.
$res = (\texttt{ok}, id, state, h; T_E)$
If $Validate(2, res; M^{id}) = \texttt{bad}$, discard.
Set $M^{id}.c1Msg = \perp$, $M^{id}.c1Time = \perp$ and $M^{id}.c1Res = res$.
(res)
(res)
If (res) has not been received within time δ^1_{on} after sending m.
$(\texttt{finalize}, 1, id)$
If $Validate(3; M^{id}) = \texttt{bad}$, discard.
Remove $M^{id}.pool[0]$ from $M^{id}.pool$ and set $M^{id}.c1Time := \perp$.
$(\texttt{bad})$

Fig. 3. Detailed executor challenge protocol.

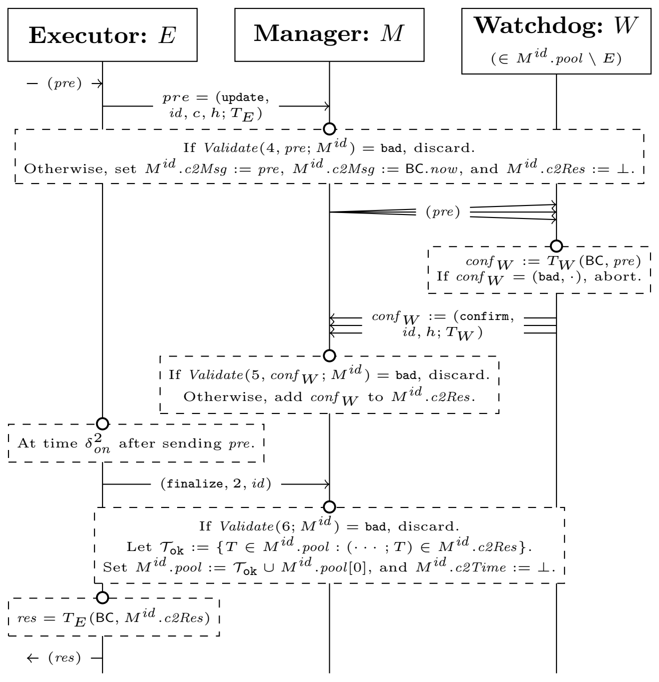
result and removes the challenge message. This finalizes the challenge procedure. If the operator does not send a valid response in time δ^1_{on}, user U sends message `finalize` to M. This triggers the manager to kick $\mathcal{E}_E$ from the execution pool of this contract and assign the next enclave 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.

5.6 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 this behavior to kick honest operators from an execution pool. This operator *denial*



Executor: E
Manager: M
Watchdog: W
($\in M^{id}.pool \setminus E$)
– (pre)
pre = (update, id, c, h; T_E)
If $Validate(4, pre; M^{id})$ = bad, discard.
Otherwise, set $M^{id}.c2Msg := pre$, $M^{id}.c2Msg :=$ BC.now, and $M^{id}.c2Res := \perp$.
(pre)
$conf_W := T_W$(BC, pre)
If $conf_W$ = (bad, ·), abort.
$conf_W$:= (confirm, id, h; T_W)
If $Validate(5, conf_W; M^{id})$ = bad, discard.
Otherwise, add $conf_W$ to $M^{id}.c2Res$.
At time δ^2_{on} after sending pre.
(finalize, 2, id)
If $Validate(6; M^{id})$ = bad, discard.
Let $\mathcal{T}_{ok} := \{T \in M^{id}.pool : (\cdots ; T) \in M^{id}.c2Res\}$.
Set $M^{id}.pool := \mathcal{T}_{ok} \cup M^{id}.pool[0]$, and $M^{id}.c2Time := \perp$.
$res = T_E$(BC, $M^{id}.c2Res$)
← (res)

Fig. 4. Detailed watchdog challenge protocol.

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 6 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.

6 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.

7 Security Analysis

In this section, we present security considerations of *POSE* based on the adversary model stated in Section 2.

7.1 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 $\epsilon-$*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 $\epsilon = 1 - \Pi_{i=0}^{s-1}(\frac{m-i}{n-i}) > 1 - (\frac{m}{n})^s$. 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.

7.2 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 7.1); 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.

8 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 area[7]. 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 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 5.5, 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

[7] `https://github.com/AppliedCryptoGroup/PoseCode`

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 449µs). 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 exist[8]. 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.

9 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

[8] https://github.com/OP-TEE/optee_os/pull/5025

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
`challangeWatchdogsCreation`	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

Table 1. 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.

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 9 "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 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 1. 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.

	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]	✗	✓	$O(1)$	✗
POSE	✓	✓	$O(1)$	✓

Table 2. Overview of related work, n is #transactions.

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.

10 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.

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 a 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).

11 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/cerfiticaten/2021/08/nscib-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. Martın 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 Hrycyszyn, 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 Stapf. 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 Stapf. 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 Kostiainen, 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 Kostiainen, Ghassan Karame, Sinisa Matetic, and Srdjan Capkun. Bitcontracts: Adding expressive smart contracts to legacy cryptocurrencies. 2019.
60. Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, 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.

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.

A.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 (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 5.4, 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.

A.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 (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 $\epsilon = 1 - \Pi_{i=0}^{s-1}(\frac{m-i}{n-i}) > 1 - (\frac{m}{n})^s$.

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

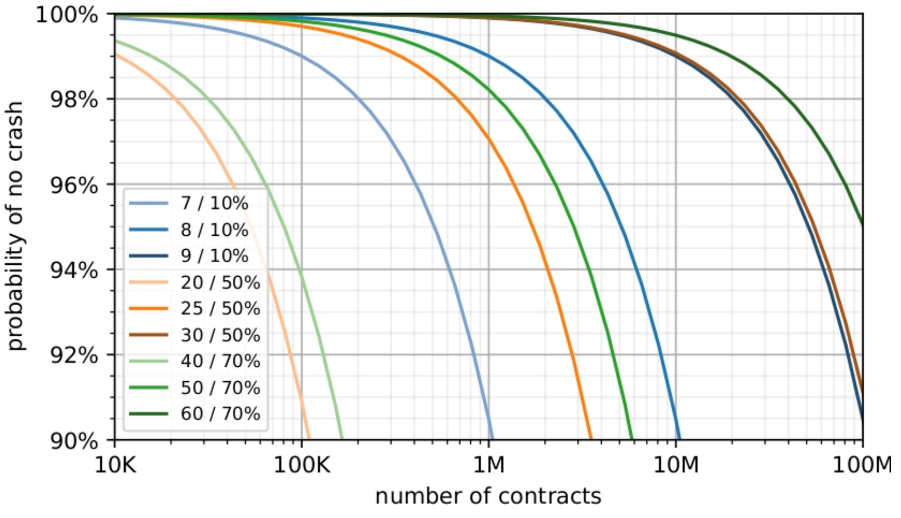


Fig. 5. Cumulative probabilities of no contracts crashing w. large number of POSE contracts for different pool sizes s and adversary shares m, "s/m".

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 $\epsilon = 1 - \Pi_{i=0}^{s-1}(\frac{m-i}{n-i}) > 1 - (\frac{m}{n})^s$. 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.

A.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 (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 2), 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 A.1) 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

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 $(\texttt{execute}, id, \cdot, \cdot; U)$. Require that $C.creator = \perp$, $C.c1Time = \perp$ and $Verify(m) = \texttt{ok}$.
- On input $(2, res; C)$, parse res to $(\texttt{ok}, id, \cdot, h; T)$. Require that $C.creator = \perp$, $H(C.c1Msg) = h$, $C.c1Time + \delta^1_{on} > \mathsf{BC}.now$, $Verify(res) = \texttt{ok}$ and $C.pool[0] = T$.
- On input $(3; C)$, require that $C \neq \perp$, $C.creator = \perp$, $Cc1Msg \neq \perp$ and $C.c1Time + \delta^1_{on} \leq \mathsf{BC}.now$.
- On input $(4, pre; C)$, parse pre to $(\texttt{update}, id, c, h; T)$. Require that $C.creator = \perp$, $C.c2Time = \perp$, $C.pool[0] = T$ and $Verify(pre) = \texttt{ok}$.
- On input $(5, conf; C)$, parse $conf$ to $(\texttt{confirm}, id, h; T_i)$ and $C.c2Msg$ to $(\cdot, \cdot, \cdot, h'; \cdot)$. Require that $C.creator = \perp$, $C.c2Time + \delta^2_{on} > \mathsf{BC}.now$, $Verify(conf) = \texttt{ok}$, $h = h'$ and $T \in C.pool$.
- On input $(6; C)$, require that $C.creator = \perp$, $C.c2Time \neq \perp$ and $C.c2Time + \delta^2_{on} \leq \mathsf{BC}.now$.

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)[9]. We recall the maximum blockchain delay which is defined as $\delta_{\mathsf{BC}} = \alpha \cdot \tau$ (cf. 2 and 4). The off-chain propagation timeout δ^2_{off} 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 $\delta^2_{on} \geq \delta^2_{off} + \delta_{\mathsf{BC}}$ 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^1_{off} 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. 4). Hence, δ^1_{off} 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 $\delta^2_{on} + \delta_{\mathsf{BC}} + \Delta$. We elaborate on maximal execution, update, and installation times of contracts in Section 5.6. Finally, $\delta^1_{on} \geq \delta^1_{off} + \delta_{\mathsf{BC}}$ 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^1_{on} 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^1_{on} timeout for the executor. Similarly, the timeout δ^1_{on} is increased by the manager if any watchdog is not responding and the executor sends a transaction that kicks this watchdog. The increased

[9] We 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.

timeout allows the executor to provide the kick transaction together with enough confirmation blocks to 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^1_{off} 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^1_{off} 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 case[10]. 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 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^1_{on}) and watchdog challenge (δ^2_{on}), 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 $(\alpha + \gamma) \cdot \tau = 1\,540s$. 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 *POSE* contract, identified by *id*, a user U sends a message $(\texttt{deposit}, id, amount; U)$ 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 $\mathit{Verify}(\texttt{deposit}, id, amount; U) = \texttt{ok}$. 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. 5.4) and the enclave forwards

[10] For 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).

the data to the *nextState* function of the contract C^{id}, U is ensure 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 $m := (\texttt{withdraw}, id, payouts, \{amount_i, U_i\}; \mathcal{E}_E)$ 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., $Verify(m) = \texttt{ok}$, $\mathcal{E}_E = M^{id}.pool[0]$, 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.

B. Putting the Online Phase on a Diet: Covert Security From Short MACs

In this chapter, we present the following publication with minor changes.

[96] 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 Faust[1], Carmit Hazay[2], David Kretzler[1], and Benjamin Schlosser[1]

[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 [28], databases [34], blockchains [2] and more [3, 1]. 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 [17, 14, 26, 27, 6, 12, 32].

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 [26, 19].

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 [5]. 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 [17], authenticated garbling [35, 36] and the TinyOT approach [31, 29, 10, 22].

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 [14, 16, 20]. 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 form 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 [31] 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., [29, 10, 22, 35, 36].

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 [5] 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. [23] 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 [15] and the MiniMac [18] protocols. The former improves the online communication complexity by relying on precomputated 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 [17]. 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. [29, 10] 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 [22].

Authenticated garbling. The authenticated garbling protocols [35, 36, 25, 37] 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 [17, 14, 26, 27, 13] provide means to perform multi-party computation with active security on arithmetic circuits. Damgård et al. [14] 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 [7, 8, 9]. 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 [5], 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 [5]. 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 [5] 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 $\epsilon^*_{\mathsf{off}}$. Our theorem states that a protocol, which is covertly secure with deterrence factor ϵ_{on} in Hy_1, is covertly secure with deterrence factor $\epsilon_{\mathsf{on}}^* := \mathsf{Min}(\epsilon_{\mathsf{on}}, \epsilon_{\mathsf{off}}^*)$ in Hy_2. While we have to impose some restrictions on the protocols that our theorem can be applied on, practical offline/online protocols [17, 31, 35, 36] 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 [31].

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 [31]. 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 constitutes 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 [5] 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 cheat_i 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 $\mathsf{corrupted}_i$ or abort_i command, which is forwarded to the honest parties. The

$\mathsf{corrupted}_i$ command models a blatant cheat option, where the adversary cheats in a way that will always be detected, and the abort_i command models an abort of a corrupted party. Later, Faust et al. [20] proposed to extract the *identifiable abort* property as it can be considered orthogonal and of independent interest (cf. [24]). For the covert notion, this means that if a corrupted party aborts, the ideal functionality only sends abort to the honest parties instead of abort_i 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 [20] 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 $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$, where $f = (f_1, \ldots, f_n)$ is the function computed by π. We define for every vector of inputs $\bar{x} = (x_1, \ldots, x_n)$ the vector of outputs $\bar{y} = (f_1(\bar{x}), \ldots, f_n(\bar{x}))$ 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 $\mathsf{REAL}_{\pi,\mathsf{Adv}(z),\mathcal{I}}(\bar{x}, 1^\kappa)$ as the output of the protocol execution π on input $\bar{x} = (x_1, \ldots, x_n)$ and security parameter κ, where Adv on auxiliary input z corrupts parties $\mathcal{I}$. We further specify $\mathsf{OUTPUT}_i(\mathsf{REAL}_{\pi,\mathsf{Adv}(z),\mathcal{I}}(\bar{x}, 1^\kappa))$ 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 $\epsilon : \mathbb{N} \to [0,1]$ 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 $\bar{x} = (x_1, \ldots, x_n)$ balanced in this case. The adversary receives an auxiliary input z. In case there is no input, the parties will receive $x_i = \mathsf{ok}$.

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 $w = \mathsf{abort}$ 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 $w = (\mathsf{cheat}_i, \epsilon_i)$ 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 $\mathsf{corrupted}_i$ to all honest parties. In addition, the ideal functionality computes $(y_1, \ldots, y_n) = f(\bar{x})$ and sends $(\mathsf{corrupted}_i, \{y_j\}_{j\in\mathcal{I}})$ to $\mathcal{S}$.
2. With probability $1 - \epsilon_i$, the ideal functionality sends $\mathsf{undetected}$ 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 $\mathsf{corrupted}_i$ for $i \in \mathcal{I}$, in which case the ideal functionality sends abort or $\mathsf{corrupted}_i$ to every P_j for $j \notin \mathcal{I}$.

The ideal execution ends at this point. Otherwise, if no w equals abort or $(\mathsf{cheat}_i, \cdot)$ the ideal execution proceeds as follows.

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

Ideal functionality answers honest parties: After receiving its outputs, the adversary sends abort, $\mathsf{corrupted}_i$ for some $i \in \mathcal{I}$, or $\mathsf{continue}$ to the ideal functionality. If the ideal functionality receives $\mathsf{continue}$ then it sends y_j to all honest parties P_j ($j \notin \mathcal{I}$). Otherwise, if it receives abort resp. $\mathsf{corrupted}_i$, it sends abort resp. $\mathsf{corrupted}_i$ 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 $\mathsf{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),\mathcal{I}}(\bar{x}, 1^{\kappa})$ 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 $\mathcal{S}$ 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 $z \in \{0,1\}^*$:*

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

The *SECF* notions follows the IECF notion with one single change. Instead of sending $(\mathsf{corrupted}_i, \{y_j\}_{j\in\mathcal{I}})$ to $\mathcal{S}$ in case of detected cheating, the ideal functionality only sends $(\mathsf{corrupted}_i)$. 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 $\epsilon_{\mathsf{off}} = 1$[3]. Then, our theorem shows that if the detection probability during the offline phase is reduced to $\epsilon'_{\mathsf{off}} < 1$, π_{on} is also covertly secure with a deterrence factor of $\epsilon'_{\mathsf{on}} = \min(\epsilon_{\mathsf{on}}, \epsilon'_{\mathsf{off}})$. 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 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 $\epsilon_{\mathsf{off}} = 1$. 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 $\epsilon'_{\mathsf{on}} = \min(\epsilon'_{\mathsf{off}}, \epsilon_{\mathsf{on}})$.

While the effect of deterrence replacement was already analyzed by Aumann and Lindell [5] 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 [5] 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 $\mathcal{F}_1, \ldots, \mathcal{F}_\ell$. 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.

[3] Covert security with deterrence factor 1 can be realized by a maliciously secure protocol as shown by Asharov and Orlandi [4].

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 $\mathcal{F}_f^\epsilon$ to denote a hybrid functionality computing function f with deterrence factor ϵ. The notation of a $(\mathcal{F}_{f_1}^{\epsilon_1}, \ldots, \mathcal{F}_{f_\ell}^{\epsilon_\ell})$-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 [11] as well as for covert security [5] 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 $\mathcal{F}_{f_{\mathsf{on}}}^{\epsilon_{\mathsf{on}}}$ in the $\mathcal{F}_{f_{\mathsf{off}}}^{1}$-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 $\epsilon'_{\mathsf{off}} \in [0, 1]$ results in a deterrence factor of the online protocol of $\epsilon'_{\mathsf{on}} = \min(\epsilon_{\mathsf{on}}, \epsilon'_{\mathsf{off}})$, 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 $f_{\mathsf{off}} : (\{\perp\}_{j \notin \mathcal{I}}, \{x_i^{\mathsf{off}}\}_{i \in \mathcal{I}}) \rightarrow (y_1^{\mathsf{off}}, \ldots, y_n^{\mathsf{off}})$ 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 $f_{\mathsf{on}} : (x_1, \ldots, x_n) \rightarrow (y_1^{\mathsf{on}}, \ldots, y_n^{\mathsf{on}})$. We use the abbreviation $\mathcal{F}_{\mathsf{off}}^{\epsilon_{\mathsf{off}}}$ and $\mathcal{F}_{\mathsf{on}}^{\epsilon_{\mathsf{on}}}$ for $\mathcal{F}_{f_{\mathsf{off}}}^{\epsilon_{\mathsf{off}}}$ and $\mathcal{F}_{f_{\mathsf{on}}}^{\epsilon_{\mathsf{on}}}$.

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 $\mathcal{F}_{\mathsf{off}}^{\epsilon}$ 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 $\mathcal{F}_{\mathsf{off}}^{\epsilon}$ returns $\mathsf{corrupted}_i$ to the parties, then π_{on} instructs the parties to output $\mathsf{corrupted}_i$. Practical offline/online protocols [17, 31, 35, 36] either directly fulfill theses 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 $\mathcal{F}_{\mathsf{on}}^{\epsilon_{\mathsf{on}}}$ in the $\mathcal{F}_{\mathsf{off}}^{1}$-hybrid model according to Definition 1, where f_{off}, f_{on} and π_{on} are defined as above. Then, π_{on} securely realizes $\mathcal{F}_{\mathsf{on}}^{\epsilon'_{\mathsf{on}}}$ in the $\mathcal{F}_{\mathsf{off}}^{\epsilon'_{\mathsf{off}}}$-hybrid model according to Definition 1, where $\epsilon'_{\mathsf{on}} = \min(\epsilon_{\mathsf{on}}, \epsilon'_{\mathsf{off}})$.*

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 [21].

On a high level, we prove our theorem by constructing a simulator $\mathcal{S}$ for the protocol π_{on} in the $\mathcal{F}_{\mathsf{off}}^{\epsilon'_{\mathsf{off}}}$-hybrid world. In the construction, we exploit the fact that π_{on} is covertly secure in the $\mathcal{F}_{\mathsf{off}}^{1}$-hybrid world with deterrence factor ϵ_{on}, which means that a simulator $\mathcal{S}_1$ for the $\mathcal{F}_{\mathsf{on}}^{\epsilon_{\mathsf{on}}}$-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 $\mathcal{F}_{\mathsf{off}}^{\epsilon'_{\mathsf{off}}}$, i.e., a set of inputs for the corrupted parties $\{x_i^{\mathsf{off}}\}_{i\in\mathcal{I}}$ together with additional input from the adversary $m \in \{\perp, \mathsf{abort}, (\mathsf{cheat}_i, \epsilon_i)\}$, where $i \in \mathcal{I}$ and $\epsilon_i \geq \epsilon'_{\mathsf{off}}$. $\mathcal{S}$ distinguishes the following cases:
   (a) If $m \in \{\perp, \mathsf{abort}\}$, $\mathcal{S}$ sends $\{x_i^{\mathsf{off}}\}_{i\in\mathcal{I}}$ 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 $\mathcal{F}_{\mathsf{on}}^{\epsilon'_{\mathsf{on}}}$ and vice versa.
   (b) If $m = (\mathsf{cheat}_\ell, \epsilon_\ell)$ for some $\ell \in \mathcal{I}$, $\mathcal{S}$ samples dummy inputs $\{\hat{x}_i^{\mathsf{on}}\}_{i\in\mathcal{I}}$ for the corrupted parties, sends $\{\hat{x}_i^{\mathsf{on}}\}_{i\in\mathcal{I}}$ together with $(\mathsf{cheat}_\ell, \epsilon_\ell)$ to $\mathcal{F}_{\mathsf{on}}^{\epsilon'_{\mathsf{on}}}$ and distinguishes the following cases:
      i. If $\mathcal{F}_{\mathsf{on}}^{\epsilon'_{\mathsf{on}}}$ replies $(\mathsf{corrupted}_i, \{\hat{y}_i^{\mathsf{on}}\}_{i\in\mathcal{I}})$, $\mathcal{S}$ computes the probabilistic function $f_{\mathsf{off}} : (\{\perp\}_{i\notin\mathcal{I}}, \{x_i^{\mathsf{off}}\}_{i\in\mathcal{I}}) \rightarrow (\hat{y}_1^{\mathsf{off}}, \ldots, \hat{y}_n^{\mathsf{off}})$ using fresh randomness, sends $(\mathsf{corrupted}_i, \{\hat{y}_i^{\mathsf{off}}\}_{i\in\mathcal{I}})$ to Adv and returns whatever Adv returns.
      ii. Otherwise, if $\mathcal{F}_{\mathsf{on}}^{\epsilon'_{\mathsf{on}}}$ replies $(\mathsf{undetected}, \{x_j^{\mathsf{on}}\}_{j\notin\mathcal{I}})$, $\mathcal{S}$ sends $\mathsf{undetected}$ to Adv and gets back the value y defined as follows:
         – If $y \in \{\mathsf{abort}, \mathsf{corrupted}_\ell\}$ for $\ell \in \mathcal{I}$, $\mathcal{S}$ sends y to $\mathcal{F}_{\mathsf{on}}^{\epsilon'_{\mathsf{on}}}$ and returns whatever Adv returns.
         – If $y = \{y_j^{\mathsf{off}}\}_{j\notin\mathcal{I}}$ with $y_j^{\mathsf{off}} \in \{0,1\}^*$ 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 $\mathcal{F}_{\mathsf{on}}^{\epsilon'_{\mathsf{on}}}$ and returns whatever Adv returns. Note that y_j^{on} can also be abort or $\mathsf{corrupted}_\ell$ for $\ell \in \mathcal{I}$.

Recall that due to first restriction on π_{on}, the call to the hybrid functionality $\mathcal{F}_{\mathsf{off}}^{\epsilon'_{\mathsf{off}}}$ 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 $\mathcal{F}_{\mathsf{off}}^{1}$-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 $\mathcal{F}_{\mathsf{off}}^{1}$-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 $\mathcal{F}_{\mathsf{off}}^{\epsilon'_{\mathsf{off}}}$-hybrid world, while it cannot happen in the $\mathcal{F}_{\mathsf{off}}^{1}$-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 $\mathcal{F}_{\mathsf{off}}^{\epsilon'_{\mathsf{off}}}$ 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 $\mathcal{F}_{\mathsf{off}}^{\epsilon'_{\mathsf{off}}}$-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 $\mathcal{F}_{\mathsf{off}}^{1}$-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 $\mathcal{F}_{\mathsf{off}}^{\epsilon'_{\mathsf{off}}}$-hybrid world is $\epsilon'_{\mathsf{on}} = \min(\epsilon_{\mathsf{on}}, \epsilon'_{\mathsf{off}})$.

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 [31]. 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 ([29, 10, 22, 36]).

Notation. We use the following notation to describe secret shared and authenticated values. This notation follows the common approach in the research field [31, 17, 35, 36]. 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 calcualte $\langle \gamma \rangle := \langle \alpha \rangle \cdot \beta$ 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 $\{1, \ldots, n\}$. We consider any sets to be ordered, e.g., $\{x_i\}_{i \in [n]} := [x_1, x_2, \ldots, x_n]$, 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 $\mathcal{F}_{\mathsf{Commit}}$ that is specified as follows:

Functionality $\mathcal{F}_{\mathsf{Commit}}$: 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 $(\mathsf{Committed}, P)$ 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 $\mathcal{F}^{\epsilon}_{f_{\mathsf{off}}}$ 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 $K[r_i] := M[r_i] \oplus r_i \cdot \Delta_B$ and $M[s_i] := K[s_i] \oplus s_i \cdot \Delta_A$. 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 $M[x] := K[x] \oplus x \cdot \Delta_A$ 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 $\mathcal{F}^{\epsilon}_{f_{\mathsf{off}}}$ in the full version of the paper [21].

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 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 [35] 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 $\mathcal{F}_{\mathsf{Commit}}$, and the hybrid covert functionality $\mathcal{F}^{1}_{f_{\mathsf{off}}}$, in the following denoted as $\mathcal{F}_{\mathsf{off}}$. 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 $\mathcal{I}_P^{\mathsf{in}}$ resp. $\mathcal{I}_P^{\mathsf{out}}$. 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 $\mathcal{F}_{\mathsf{off}}$ 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 $\mathcal{F}_{\mathsf{off}}$, returns $m \in \{\mathsf{abort}, \mathsf{corrupted}_{\bar{P}}\}$, 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 $i := \mathcal{I}_P^{\mathsf{in}}[j]$, 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 $\mathcal{M}^1_{(P,P)} := H(\mathcal{M}_P^P)$ and $\mathcal{M}^1_{(P,\bar{P})} = H(\mathcal{M}_{\bar{P}}^P)$ and sends $\mathcal{M}^1_{(P,P)}$.
5. Each party $P \in \{\mathcal{A}, \mathcal{B}\}$, upon receiving $\mathcal{M}^1_{(\bar{P},\bar{P})}$, verifies that $\mathcal{M}^1_{(\bar{P},\bar{P})} = \mathcal{M}^1_{(P,\bar{P})}$. If not, P outputs $\mathsf{corrupted}_{\bar{P}}$ and aborts. Otherwise, P computes $\mathcal{M}^2_{(P,P)} := H(\{M[w_i^P]\}_{i \in \mathcal{I}_{\bar{P}}^{\mathsf{out}}})$, and sends $(\mathsf{Commit}, (\{w_i^P\}_{i \in \mathcal{I}_{\bar{P}}^{\mathsf{out}}}, \mathcal{M}^2_{(P,P)}))$ to $\mathcal{F}_{\mathsf{Commit}}$.
6. Upon receiving, $(\mathsf{Committed}, \bar{P})$ from $\mathcal{F}_{\mathsf{Commit}}$, P sends (Open) to $\mathcal{F}_{\mathsf{Commit}}$.
7. Each party $P \in \{\mathcal{A}, \mathcal{B}\}$, upon receiving $(\mathsf{Opened}, \bar{P}, (\{w_i^{\bar{P}}\}_{i \in \mathcal{I}_P^{\mathsf{out}}}, \mathcal{M}^2_{(\bar{P},\bar{P})}))$ from $\mathcal{F}_{\mathsf{Commit}}$, re-defines $\mathcal{M}_{\bar{P}}^P := \{K[w_i^{\bar{P}}] \oplus w_i^{\bar{P}} \cdot \Delta_P\}_{i \in \mathcal{I}_P^{\mathsf{out}}}$ and verifies that $\mathcal{M}^2_{(\bar{P},\bar{P})} = H(\mathcal{M}_{\bar{P}}^P)$. If not, P outputs $\mathsf{corrupted}_{\bar{P}}$ and aborts. Otherwise, P outputs $\{w_i^P \oplus w_i^{\bar{P}}\}_{i \in \mathcal{I}_P^{\mathsf{out}}}$.

Handle aborts:

8. If a party P does not receive a timely message before executing Step 6, it outs abort and aborts. If a party P does not receive a timely message after having executed Step 6, it outputs $\mathsf{corrupted}_{\bar{P}}$ 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 $M[b^{\mathcal{A}}]$ for a bit $b^{\mathcal{A}}$ known to $\mathcal{A}$ is defined as $K[b^{\mathcal{A}}] \oplus b^{\mathcal{A}} \cdot \Delta_{\mathcal{B}}$, a MAC $\widetilde{M}[\tilde{b}^{\mathcal{A}}]$ of a flipped bit $\tilde{b}^{\mathcal{A}}$ is correct iff $\widetilde{M}[\tilde{b}^{\mathcal{A}}] = M[b^{\mathcal{A}}] \oplus \Delta_{\mathcal{B}} = K[b^{\mathcal{A}}] \oplus (b^{\mathcal{A}} \oplus 1) \cdot \Delta_{\mathcal{B}}$. 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 [21]:

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 $\mathcal{F}_f^{\epsilon}$ (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 $(\mathcal{F}_{\mathsf{off}}, \mathcal{F}_{\mathsf{Commit}})$-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 a 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 [16, 20, 33]. 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 [21]. 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

$$(t+1) \cdot C_{\mathsf{Commit}} + t \cdot (C_{\mathsf{OT}} + C_{\mathsf{Open}} + n_2 \cdot (3 + \kappa - 1) + (n_1 + 2 \cdot n_2) \cdot (M - 1))$$

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 $\kappa = 128$, $\lambda = 40$, $C_{\mathsf{OT}} = (2 + \kappa) \cdot 256$ according to [30], $C_{\mathsf{Commit}} = 256$ and $C_{\mathsf{Open}} = 2 \cdot \kappa$ 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.

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 $\mathcal{F}^1_{\mathsf{off}}$-hybrid world.

First, the hybrid functionality $\mathcal{F}_{\mathsf{off}}$ 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 $\mathcal{F}_{\mathsf{off}}$, 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 $\mathcal{F}_{\mathsf{off}}$, $\mathcal{S}$ would have to decide whether it simulates this interactions itself or via $\mathcal{S}_1$. This means that the adversary's input to $\mathcal{F}_{\mathsf{off}}$ 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 $\mathcal{F}_{\mathsf{off}}$ can influence the adversary's input to $\mathcal{F}_{\mathsf{off}}$, and hence, $\mathcal{S}$'s decision to simulate the steps afterwards based on $\mathcal{S}_1$ or not.

Second, we require that in case $\mathcal{F}_{\mathsf{off}}$ outputs $\mathsf{corrupted}$, the protocol π_{on} instructs the parties to output $\mathsf{corrupted}$ 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 $\mathcal{F}^{\epsilon_{\mathsf{on}}}_{\mathsf{on}}$ about the cheating attempt in the offline phase. In case of detected cheating, $\mathcal{F}^{\epsilon'_{\mathsf{on}}}_{\mathsf{on}}$ sends $\mathsf{corrupted}$ to the honest parties and thus the honest parties output $\mathsf{corrupted}$ 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 $\mathsf{corrupted}$ in the real world, too.

Finally, we emphasize that known offline/online protocols (SPDZ [17], TinyOT [31], authenticated garbling [35, 36]) either directly fulfill the aforementioned requirements or can easily be adapted to do so.

B Comparison of Theorem 1 with [5]

Aumann and Lindell [5] presented a sequential composition theorem for the (strong) explicit cheat formulation. The theorem shows that a protocol π that is covertly secure in an $(\mathcal{F}^{\epsilon_1}_1, \ldots, \mathcal{F}^{\epsilon_{p(n)}}_{p(n)})$-hybrid world with deterrence factor ϵ_π, i.e., parties have access to a polynomial number of functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_{p(n)}$ with deterrence factor $\epsilon_1, \ldots, \epsilon_{p(n)}$, 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 $i \in \{1, \ldots, p(n)\}$. 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.

Out theorem takes on 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

1. MPC Alliance. https://www.mpcalliance.org/. (Accessed on 10/14/2022).
2. ZenGo - crypto wallet app. https://zengo.com/. (Accessed on 10/14/2022).
3. 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.
4. Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In *ASIACRYPT*, 2012.
5. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, 2007.
6. Carsten Baum, Daniele Cozzo, and Nigel P. Smart. Using topgear in overdrive: A more efficient zkpok for SPDZ. In *SAC*, 2019.
7. 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.
8. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density LPN. In *FOCS*, 2020.
9. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *CRYPTO*, 2020.
10. 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.
11. Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1), 2000.
12. 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.
13. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spd$\mathcal{F}_{2^k}$: 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.
14. 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.

15. 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.
16. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. In *CRYPTO*, 2020.
17. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.
18. Ivan Damgård and Sarah Zakarias. Constant-overhead secure computation of boolean circuits using preprocessing. In *TCC*, 2013.
19. Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Authenticated garbling from simple correlations. In *CRYPTO*, 2022.
20. Sebastian Faust, Carmit Hazay, David Kretzler, and Benjamin Schlosser. Generic compiler for publicly verifiable covert multi-party computation. In *EUROCRYPT*, 2021.
21. 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.
22. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *ASIACRYPT*, 2015.
23. 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.
24. Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO*, 2014.
25. Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In *CRYPTO*, 2018.
26. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.
27. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, 2018.
28. 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.
29. Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In *CRYPTO*, 2014.
30. Ian McQuoid, Mike Rosulek, and Lawrence Roy. Batching base oblivious transfers. In *ASIACRYPT*, 2021.
31. 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.
32. Emmanuela Orsini. Efficient, actively secure MPC with a dishonest majority: A survey. In *WAIFI*, 2020.
33. Peter Scholl, Mark Simkin, and Luisa Siniscalchi. Multiparty computation with covert security and public verifiability. In *ITC*, 2022.
34. Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In *EuroSys*, 2019.
35. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, 2017.
36. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.
37. Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In *CCS*, 2020.

C. Generic Compiler for Publicly Verifiable Covert Multiparty Computation

In this chapter, we present the following publication with minor changes.

[95] 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 Faust[1], Carmit Hazay[2], David Kretzler[1], and Benjamin Schlosser[1]

[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 strengthen 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 $\frac{s}{\log|C|}$ [26], where C is the computed circuit and s is a statistical security parameter. For secret sharing-based protocols, Hazay et al. [15] have recently shown a constant communication overhead over the semi-honest GMW-protocol [13]. 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 [25, 26] or the generation of correlated randomness in form of Beaver triples [4] in secret sharing-based protocols such as in SPDZ [11]. 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 [25, 26] and [11].

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 [3], 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. [14] and later been adapted to a different line of MPC protocols by Damgård et al. [9].

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 [2], 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 [29], 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 [2] most works have explored publicly verifiable covert security in the two-party setting [19, 16, 29, 10]. 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 [14, 8, 20, 9]. 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. [10]. While [10] 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 [10] achieves $\approx \frac{t-1}{nt}$, which in particular for a large number of parties n results in a low deterrence factor. [10] 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 [10] 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. [10] 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 [28] and [11], 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* [24, 22, 21, 5], 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 [6]. 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 [24, 22] 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. [10] 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 [18]). Next, it obliviously 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 [7] or Jacobians of hyperelliptic curves [12]. 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 $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$, where $f = (f_1, \dots, f_n)$, be the function realized by Π. 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 $\mathsf{REAL}_{\Pi,\mathcal{A}(z),I}(\bar{x}, 1^\kappa)$ 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 $\mathsf{OUTPUT}_i(\mathsf{REAL}_{\Pi,\mathcal{A}(z),I}(\bar{x}, 1^\kappa))$, 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 [3]. 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

[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 [1].

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 [3]. 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 $w_i = \mathsf{corrupted}_i$ to the trusted party, then the trusted party sends $\mathsf{corrupted}_i$ to all of the honest parties and halts. If multiple parties send $\mathsf{corrupted}_i$, then the trusted party disregards all but one of them (say, the one with the smallest index i). If both $\mathsf{corrupted}_i$ and abort messages are sent, then the trusted party ignores the $\mathsf{corrupted}_i$ message.

Attempted cheat option: If a corrupted party sends additional input $w_i = \mathsf{cheat}_i$ to the trusted party (as above: if there are several messages $w_i = \mathsf{cheat}_i$ 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 $\mathsf{corrupted}_i$ to the adversary and all of the honest parties.
2. With probability $1 - \epsilon$, the trusted party sends $\mathsf{undetected}$ 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 abort_i, $\mathsf{corrupted}_i$ or cheat_i, the ideal execution continues below.

Trusted party answers honest parties: If the trusted party did not receive $\mathsf{corrupted}_i$, cheat_i 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 $\mathsf{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),I}(\bar{x}, 1^{\kappa})$ 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 $z \in \{0,1\}^*$:

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

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* [17], 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 [9] 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. [10].

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 $\mathcal{F}_{\mathsf{Cov}}$.

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 [2]. 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 $\mathsf{corrupted}_j$ in the protocol execution for $j \in I$ and returns a certificate Cert, i.e., $\mathsf{Cert} := \mathsf{Blame}(\mathsf{view}_i)$. 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 $\mathsf{Cert} = \mathsf{Blame}(\mathsf{view}_i)$ 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 $(\Pi', \mathsf{Blame}, \mathsf{Judge})$* 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 $\mathsf{OUTPUT}_j(\mathsf{REAL}_{\Pi,\mathcal{A}(z),I}(\bar{x}, 1^\kappa)) = \mathsf{corrupted}_i$ for $j \in [n] \setminus I$ and $i \in I$ then:*
$$\Pr[\mathsf{Judge}(\mathsf{Cert}) = \mathsf{id}_i] > 1 - \mu(n),$$
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 $j \in [n] \setminus I$:*
$$\Pr[\mathsf{Cert}^* \leftarrow \mathcal{A}; \mathsf{Judge}(\mathsf{Cert}^*) = \mathsf{id}_j] < \mu(n).$$

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 [27]. 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 $\delta_{\mathsf{Sq}}(\kappa) = T(\mathcal{S}, (N, g))$, 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. [24] 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. [6] 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* ($\mathsf{TL.Setup}$, $\mathsf{TL.Generate}$, $\mathsf{TL.Solve}$, $\mathsf{TL.Verify}$) *defined as follows.*

– $(pp) \leftarrow \mathsf{TL.Setup}(1^\lambda, \mathcal{T})$ *takes as input the security parameter* 1^λ *and a hardness parameter* $\mathcal{T}$*, and outputs public parameter* pp*.*
– $p \leftarrow \mathsf{TL.Generate}(pp, s)$ *takes as input public parameters* pp *and a solution* $s \in \mathcal{S}$ *and outputs a puzzle* p*.*
– $(s, \pi) \leftarrow \mathsf{TL.Solve}(pp, p)$ *is a deterministic algorithm that takes as input public parameters* pp *and a puzzle* p *and outputs a solution* s *and a proof* π*.*
– $b := \mathsf{TL.Verify}(pp, p, s, \pi)$ *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* $\mathsf{TL.Verify}$ *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* $pp \leftarrow \mathsf{TL.Setup}(1^\lambda, \mathcal{T})$*, and for all* s*, it holds that*

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

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

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

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

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

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*

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

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 [24]. 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 [7] or Jacobians of hyperelliptic curves [12]. 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 [27] 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 [23] 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 $\mathsf{Primes}(2k)$. The set $\mathsf{Primes}(2k)$ 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 know. 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 $\mathcal{T}' := \mathcal{T}/\delta_{\mathsf{Sq}}(\lambda)$.
- Sample uniform $\tilde{g} \xleftarrow{\$} \mathbb{Z}_N^*$ and set $g := -\tilde{g}^2(\mod N)$.
- 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 $r \xleftarrow{\$} \{1, \ldots, N^2\}$.
- Compute $g^* := g^r$ and $h^* := h^r$.
- Set $c^* := h^* \cdot s \mod N$.
- Output $p := (g^*, c^*)$.

TL.Solve(pp, p):

- Parse $pp := (\mathcal{T}', N, Z)$ and $p := (g^*, c^*)$.
- Compute $h := {g^*}^{2^{\mathcal{T}'}}(\mod N)$ by repeated squaring.
- Compute $s := \frac{c^*}{h} \mod N$ as the solution.
- Compute $\ell = H_{\mathsf{prime}}(\mathsf{bin}(g^*) || \star || \mathsf{bin}(s)) \in \mathsf{Primes}(2k)$ 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 $\ell = H_{\mathsf{prime}}(\mathsf{bin}(g^*)||\star||\mathsf{bin}(s)) \in \mathsf{Primes}(2k)$ as the challenge.
- Compute $r = 2^{\mathcal{T}'} \mod \ell$.
- 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 $\mathsf{Primes}(2k)$ and the factorization of N is unknown. See [27] 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 $(c, d) \leftarrow \mathsf{Commit}(m)$ to commit to message m, where c is the commitment value and d denotes the decommitment or opening value. Similarly, we use $m' \leftarrow \mathsf{Open}(c, d)$ 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 $\mathsf{Commit}(x) = H(i, x, r)$ where i is the identity of the committer, $H : \{0,1\}^* \to \{0,1\}^{2\kappa}$ is a random oracle and $r \overset{\$}{\leftarrow} \{0,1\}^{\kappa}$.

3.4 Signature Scheme

We use a signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ 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 $(\mathsf{pk}, \mathsf{sk})$. 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 $(y^1, \ldots, y^n) \leftarrow f()$, 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 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.

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

3.6 Coin Tossing Functionality

We utilize a maliciously secure coin tossing functionality $\mathcal{F}_{\mathsf{coin}}$ 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 $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\kappa$ to all parties.

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

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

- Sample $\mathsf{seed} \xleftarrow{\$} \{0,1\}^\kappa$.
- 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 $\mathcal{F}_{\mathsf{PG}}$ 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. $\mathcal{F}_{\mathsf{PG}}$ 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 expect 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 $\mathcal{F}_{\mathsf{PG}}$ can be instantiated with a general purpose maliciously secure MPC-protocol such as the ones specified by [9] or [28].

Functionality $\mathcal{F}_{\mathsf{PG}}$

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 $r := \sum_{i=1}^n r^i \mod t$ and $u := \bigoplus_{i=1}^n u^i$.
- Generate puzzle $p \leftarrow \mathsf{TL.Generate}(pp, (r, \{d_j^i\}_{i\in[n], j\in[t]\setminus r}))$ 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 $\mathsf{continue}$ 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 $\mathsf{continue}$, 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

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

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 $\mathsf{seed}^{(1,h)}$. Additionally, P_h and all other parties get uniform κ-bit strings $\{\mathsf{seed}^{(2,i)}\}_{i\in[n]}$, which are the public seed shares of all parties. The randomness used by P_h in the semi-honest protocol will be derived from $\mathsf{seed}^h := \mathsf{seed}^{(1,h)} \oplus \mathsf{seed}^{(2,h)}$. 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 $\mathsf{seed}^{(1,h)}$, sets $(c^h, d^h) \leftarrow \mathsf{Commit}(\mathsf{seed}^{(1,h)})$, and sends c^h to all parties.

(b) **Public coin-phase**
For each $i \in [n]$, party P_h sends ok to $\mathcal{F}_{\mathsf{coin}}$ and receives $\mathsf{seed}^{(2,i)}$.
Output
If P_h has not received all messages in the expected communication rounds or any $\mathsf{seed}^{(2,i)} = \perp$, it sends abort to all parties and outputs abort.
Otherwise, it outputs $(\mathsf{seed}^{(1,h)}, d^h, \{\mathsf{seed}^{(2,i)}, c^i\}_{i\in[n]})$.

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 $(\mathsf{sk}, \mathsf{pk})$ and all parties know the public keys of the other parties. Furthermore, we suppose the setup of the verifiable time-lock puzzle scheme $\mathsf{TL.Setup}$ 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 $\mathsf{TL.Setup}$-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 $\mathcal{F}_{\mathsf{PG}}$ 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 $\mathcal{F}_{\mathsf{PG}}$ and can verify that opening the commitments is successful. In the pessimistic case, solving the time-lock puzzle reveals the input to $\mathcal{F}_{\mathsf{PG}}$. 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 κ, n, t, Π_{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

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

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

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:

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

Create publicly verifiable evidence:

3. Party P_h samples a coin share $r^h \overset{\$}{\leftarrow} [t]$, a randomness share $u^h \overset{\$}{\leftarrow} \{0,1\}^\kappa$, sends the message $(r^h, u^h, \{d_j^h\}_{j\in[t]})$ to $\mathcal{F}_{\mathsf{PG}}$ and receives time-lock puzzle p as response.
4. For each $j \in [t]$, Party P_h creates a signature $\sigma_j^h \leftarrow \mathsf{Sign}_{\mathsf{sk}_h}(\mathsf{data}_j)$, where the signed data is defined as
$$\mathsf{data}_j := (h, j, \{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}, \{c_j^i\}_{i\in[n]}, p, \mathsf{trans}_j).$$
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.
   - $\mathcal{F}_{\mathsf{PG}}$ returned abort, or
   - any other party has sent abort

   party P_h broadcasts and outputs abort.
6. Otherwise, P_h sends $\mathsf{continue}_h$ to $\mathcal{F}_{\mathsf{PG}}$, receives $(r, \{d_j^{*i}\}_{i\in[n], j\in[t]\setminus r})$ as response and calculates
$$(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$$
where view^h is the view of P_h.
If $\mathsf{cert} \neq \perp$, broadcast cert and output $\mathsf{corrupted}_m$. Otherwise, P_h outputs y_r^h.

Pessimistic case:

7. If $\mathcal{F}_{\mathsf{PG}}$ returned abort in step 6, P_h solves the time-lock puzzle
$$((r, \{d_j^{*i}\}_{i\in[n], j\in[t]\setminus r}), \pi) := \mathsf{TL.Solve}(pp, p)$$
and calculates
$$(m, \mathsf{cert}) := \mathsf{Blame}(\mathsf{view}^h)$$
where view^h is the view of P_h.
If $\mathsf{cert} \neq \perp$, broadcast cert and output $\mathsf{corrupted}_m$. 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 $\{\mathsf{seed}_j^{(2,i)}\}_{i\in[n]}$
- 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
  $(\{\mathsf{pk}_j\}_{i\in[n]}, \{\mathsf{data}_j\}_{j\in[t]}, \pi, \{\sigma_j^i\}_{i\in[n], j\in[t]})$

do:

1. Calculate $\mathsf{seed}_j^{(1,i)} := \mathsf{Open}(c_j^i, d_j^i)$ for each $i \in [n], j \in [t] \setminus r$.
2. Let $M_1 := \{(i,j) \in ([n],[t] \setminus r) : \mathsf{seed}_j^{(1,i)} = \perp\}$. If $M_1 \neq \emptyset$, choose the tuple $(m,l) \in M_1$ with minimal m and l, prioritized by m, compute $(\cdot, \pi) := \mathsf{TL.Solve}(pp, p)$, if $\pi = \perp$, set $\mathsf{cert} := (\mathsf{pk}_m, \mathsf{data}_j, \pi, r, \{d_j^i\}_{i\in[n], j\in[t]\setminus r}, \sigma_l^m)$ and output (m, cert).
3. Set $\mathsf{seed}_j^i := \mathsf{seed}_j^{(1,i)} \oplus \mathsf{seed}_j^{(2,i)}$ 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 $(\mathsf{trans}_j', \cdot)$.
5. Let $M_2 := \{j \in [t] \setminus r : \mathsf{trans}_j' \neq \mathsf{trans}_j\}$. 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 $\mathsf{cert} := (\mathsf{pk}_m, \mathsf{data}_l, \{d_l^i\}_{i\in[n]}, \sigma_l^m)$ 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 $\mathsf{cert} = (\mathsf{pk}_m, \mathsf{data}, \pi, r, \{d_j^i\}_{i\in[n], j\in[t]\setminus r}, \sigma_l^m)$ do:

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

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

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

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.

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 $\mathcal{F}_{\mathsf{PG}}$ 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 $\mathcal{F}_{\mathsf{PG}}$ 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, expect 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.

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

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 $\{c_l^i\}_{i \in [n]}$, the public seeds $\{\mathsf{seed}^{(2,i)}\}_{i \in [n]}$, and the certificate contains the valid openings $\{d_l^i\}_{i \in [n]}$, 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 $\{d_l^i\}_{i \in [n]}$ 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 signatures 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 $\mathsf{Open}(c_j^{*h}, d_j^{*h}) = \perp$ 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 $\mathcal{F}_{\mathsf{PG}}$. 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^* or seed_j^h needs to be incorrect. Also, P_h does not create a signature for an invalid trans^*. Thus, trans^* is either correct or the adversary can forge signatures. The seed_j^h is calculated as $\mathsf{seed}_j^h := \mathsf{seed}_j^{(1,h)} \oplus \mathsf{seed}_j^{(2,h)}$. The public seed $\mathsf{seed}_j^{(2,h)}$ 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 $\mathsf{seed}_j^{(2,h)}$ 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 $(\mathsf{trans}_j^*, c_j^h, d_j^h, \mathsf{seed}_j^{(2,h)})$ 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., [3, 2, 10]). 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., [10], 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 particular 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 [1] 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 $u = \Sigma\ u_i$, 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. [10] 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. [10], 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 $\mathsf{Pr}[\mathsf{bad}]$ 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 $\mathsf{Pr}[\mathsf{bad}]$ and the expected number of protocol runs runs are calculated based on the inclusion-exclusion principle as follows:

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

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.

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.

n	t	**Our approach**	**Watchlist approach**		
		ϵ	ϵ'	**or**	runs
2	2	1/2	-		2
	3	2/3	1/3		3
	10	9/10	4/10		10
3	2	1/2	-		4
	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.

References

1. Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 430–454. Springer, Heidelberg, April 2015.
2. Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 681–698. Springer, Heidelberg, December 2012.
3. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 137–156. Springer, Heidelberg, February 2007.
4. Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992.
5. Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. Time-lock puzzles from randomized encodings. In *ITCS 2016*, 2016.
6. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, August 19–23, 2018.
7. Johannes Buchmann and Hugh C. Williams. A key-exchange system based on imaginary quadratic fields. *Journal of Cryptology*, 1(2):107–118, June 1988.
8. Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. From passive to covert security at low cost. In Daniele Micciancio, editor, *TCC 2010*, volume 5978 of *LNCS*, pages 128–145. Springer, Heidelberg, February 2010.
9. 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 Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.
10. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 647–676. Springer, 2020.
11. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.
12. Samuel Dobson and Steven D. Galbraith. Trustless groups of unknown order with hyperelliptic curves. *IACR Cryptol. ePrint Arch. 2020*, 2020:196, 2020.

13. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th ACM STOC 1987*, pages 218–229, 1987.
14. Vipul Goyal, Payman Mohassel, and Adam Smith. Efficient two party and multi party computation against covert adversaries. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 289–306. Springer, Heidelberg, April 2008.
15. Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, and Mor Weiss. The price of active security in cryptographic protocols. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 184–215. Springer, 2020.
16. Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wen-jie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. In *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 97–121. Springer, Heidelberg, 2019.
17. Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 369–386. Springer, Heidelberg, August 2014.
18. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer - efficiently. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 572–591. Springer, Heidelberg, August 2008.
19. Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 210–235. Springer, Heidelberg, November / December 2015.
20. Yehuda Lindell, Eli Oxman, and Benny Pinkas. The IPS compiler: Optimizations, variants and concrete efficiency. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 259–276. Springer, Heidelberg, August 2011.
21. Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Time-lock puzzles in the random oracle model. In *CRYPTO 2011*, 2011.
22. Giulio Malavolta and Sri Aravinda Krishnan Thyagarajan. Homomorphic time-lock puzzles and applications. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 620–649. Springer, 2019.
23. Krzysztof Pietrzak. Simple verifiable delay functions. In *ITCS 2019*, volume 124, pages 60:1–60:15. LIPIcs, January 2019.
24. 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.
25. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM CCS 17*, pages 21–37. ACM Press, 2017.
26. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *ACM CCS 17*, pages 39–56. ACM Press, 2017.
27. Benjamin Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 379–407. Springer, Heidelberg, 2019.
28. Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1627–1646. ACM, 2020.
29. Ruiyu Zhu, Changchang Ding, and Yan Huang. Efficient publicly verifiable 2pc over a blockchain with applications to financially-secure computations. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 633–650. ACM, 2019.

D. Financially Backed Covert Security

In this chapter, we present the following publication with minor changes.

[94] 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 Faust[1], Carmit Hazay[2], David Kretzler[1], and Benjamin Schlosser[1]

[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* [3]. 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 [2] 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 [2, 17, 25, 15]. Recently, Damgård et al. [9] 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. [13] and Scholl et al. [19].

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 straightforward 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., [18, 5, 1]) 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. [25], 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. [25] 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 [25] to work for other protocol types and the multi-party case requires us to address several challenges. First, in [25] 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., [14, 4]) 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 [25] 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. [25], 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 [25] 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 [25] 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 [9, 13, 19], 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 on 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 $(\pi_{\mathsf{cov}}, \mathsf{Blame}, \mathsf{Judge})$ 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 $(\pi'_{\mathsf{cov}}, \mathsf{Blame}', \mathsf{Punish})$ 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 [23], 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. [6] and subsequently by many further constructions [16, 20, 25, 11]. 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 reference 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 $\mathsf{negl}(n) : \mathbb{N} \to \mathbb{R}$ is *negligible* in n if for every positive integer c there exists an integer n_0 such that $\forall n > n_0$ it hols that $\mathsf{negl}(n) < \frac{1}{n^c}$. We use the notation $\mathsf{negl}(n)$ to denote a negligible function.

We define $\mathsf{REAL}_{\pi,\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa)$ 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 $\mathsf{OUTPUT}_j(\mathsf{REAL}_{\pi,\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa))$ to be the output of party P_j for $j \in [n]$.

Our protocol utilizes a signature scheme $(\mathsf{Generate}, \mathsf{Sign}, \mathsf{Verify})$ that is *existentially unforgeable under chosen-message attacks.* We assume that each party executes the $\mathsf{Generate}$-algorithm to obtain a key pair $(\mathsf{pk}, \mathsf{sk})$ 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 $(x, \sigma := \mathsf{Sign}_{\mathsf{sk}_i}(x))$. The verification is therefore written as $\mathsf{Verify}(\langle x \rangle_i)$ instead of $\mathsf{Verify}_{\mathsf{pk}_i}(x, \sigma)$. Further, we make use of a hash function $H(\cdot) : \{0,1\}^* \to \{0,1\}^\kappa$ 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 $\mathsf{msg}_{(\ell,k)}^{(i,j)}$. The hash of this message is denoted with $\mathsf{hash}_{(\ell,k)}^{(i,j)} := H(\mathsf{msg}_{(\ell,k)}^{(i,j)})$.

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 $H(\cdot) : \{0,1\}^* \to \{0,1\}^\kappa$.

Formally, we define a Merkle tree as a tuple of algorithms $(\mathsf{MTree}, \mathsf{MRoot}, \mathsf{MProof}, \mathsf{MVerify})$. 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 $\mathsf{MVerify}$ takes as input a proof σ, an index i, a root root and a leaf x^* and returns true iff x^* is the i-the 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 $j \in [N]$: $\mathsf{MVerify}(\mathsf{MProof}(x_j, \mathsf{mTree}), j, \mathsf{MRoot}(\mathsf{mTree}), x_j) = \mathsf{true}$. We call this property *correctness*. Second, for each Merkle tree mTree with root $\mathsf{root} := \mathsf{MRoot}(\mathsf{mTree})$ 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 $\mathsf{MVerify}(\sigma^*, j^*, \mathsf{MRoot}(\mathsf{mTree}), x^*) = \mathsf{true}$ is $\mathsf{negl}(\kappa)$. 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., $b_i^{(\mathsf{pre})}$, and after the execution of the protocol Punish, i.e., $b_i^{(\mathsf{post})}$. 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 withhold 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* $(\pi, \mathsf{Blame}, \mathsf{Punish})$ *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 $\mathsf{OUTPUT}_h(\mathsf{REAL}_{\pi,\mathcal{A}(z),\mathcal{I}}(\bar{x}, 1^\kappa)) = \mathsf{corrupted}_*$ [3], then $\exists\, m \in \mathcal{I}$ such that:
$$\Pr[b_m^{(\mathsf{post})} = b_m^{(\mathsf{pre})} - d] > 1 - \mu(\kappa),$$
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:*
$$\Pr[b_j^{(\mathsf{post})} < b_j^{(\mathsf{pre})}] < \mu(\kappa).$$

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 follows 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, $\mathsf{Exp}^{\mathsf{FA}}$ and $\mathsf{Exp}^{\mathsf{FDF}}$, 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.

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 $\{\mathsf{cert}_i\}_{i \in [n] \setminus \mathcal{I}}$ 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 $\{\mathsf{view}_i\}_{i \in \mathcal{I}}$. 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

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

be addressed by the adversary separately and are all executed by the challenger: $\{\mathtt{P_h}\}_{\mathtt{h}\in[n]\setminus\mathcal{I}}$, $\mathtt{J}$, and $\mathtt{L}$. 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 $\mathsf{corrupted}_m$ 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 $\{b_i^{(\mathsf{prePunish})}\}_{i\in[n]}$. Note that $\mathsf{prePunish}$ 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., $b_i^{\mathsf{prePunish}} = b_i^{\mathsf{pre}} - d$. After the execution, the challenger $\mathcal{C}$ again reads the balances of all parties storing them as $\{b_i^{(\mathsf{post})}\}_{i\in[n]}$. If $b_m^{(\mathsf{post})} = b_m^{(\mathsf{prePunish})} + d$ 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 $\mathsf{Exp}^{\mathsf{FA}}$ is at most negligible, i.e., if $\Pr[\mathsf{Exp}^{\mathsf{FA}}(\mathcal{A},\kappa) = 1] \leq \mathsf{negl}(\kappa)$.

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 $b_h^{(\mathsf{post})} < b_h^{(\mathsf{prePunish})} + d$ 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 $\mathsf{Exp}^{\mathsf{FDF}}$ is at most negligible, i.e. if $\Pr[\mathsf{Exp}^{\mathsf{FDF}}(\mathcal{A},\kappa) = 1] \leq \mathsf{negl}(\kappa)$.

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 [9, 13, 19] 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 $\mathsf{computeRound}_k^i$ that on input the previous internal state $\mathsf{state}_{(k-1)}^{(i)}$ and the set of incoming messages $\{\mathsf{msg}_{(k-1)}^{(j,i)}\}_{j \neq i}$ computes the new internal state $\mathsf{state}_{(k)}^{(i)}$ and the set of outgoing messages $\{\mathsf{msg}_{(k)}^{(i,j)}\}_{j \neq i}$. 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

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

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 $\mathsf{emulate}_\pi^{\mathsf{full}}$ emulates all parties while the second algorithm $\mathsf{emulate}_\pi^{\mathsf{part}}$ emulates only a partial subset of the parties and considers the messages of all other parties as correct. We formally define them as

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

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 [9] resp. [13, 19]. We elaborate on these protocols in the full version of this paper.

To model this behavior formally, we define the abstract tuples $\mathsf{initData}^{\mathsf{core}}$ and $\mathsf{initData}^{\mathsf{aux}}$ as well as the algorithm $\mathsf{deriveInit}$. $\mathsf{initData}_{(i)}^{\mathsf{core}}$ 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). $\mathsf{initData}_{(i)}^{\mathsf{aux}}$ represents the additional data all parties receive during the PVC protocol that can be used to interpret $\mathsf{initData}_{(i)}^{\mathsf{core}}$ (e.g., the verifiable solution of the time-lock puzzle). Finally, $\mathsf{deriveInit}$ is an algorithm that on input $\mathsf{initData}_{(i)}^{\mathsf{core}}$ and $\mathsf{initData}_{(i)}^{\mathsf{aux}}$ derives the initial state of party P_i (e.g., verifying the solution of the puzzle). Instead of outputting an initial state, the algorithm $\mathsf{deriveInit}$ can also output bad or $\bot$. The former states that party P_i misbehaved during the PVC protocol by providing inconsistent data. The symbol $\bot$ states that the input to $\mathsf{deriveInit}$ has been invalid which can only occur if $\mathsf{initData}_{(i)}^{\mathsf{core}}$ or $\mathsf{initData}_{(i)}^{\mathsf{aux}}$ 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 $\mathsf{deriveInit}(x, y_1) \neq \bot$, $\mathsf{deriveInit}(x, y_2) \neq \bot$, and $\mathsf{deriveInit}(x, y_1) \neq \mathsf{deriveInit}(x, y_2)$ is negligible. The hiding property requires that the probability of a polynomial time adversary finding for a given $\mathsf{initData}^{\mathsf{core}}$ a $\mathsf{initData}^{\mathsf{aux}}$ such that $\mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}, \mathsf{initData}^{\mathsf{aux}}) \neq \bot$ 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 [13] 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 [9, 13] or a common public transcript of hashes [19]. However, [9] and [13] 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 $\mathsf{state}^{(j)}_{(0)}$. 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 $\mathsf{state}^{(j)}_{(k)}$ 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., $H(\mathsf{state}^{(i)}_{(\ell,k)})$. At the end of a protocol instance each party P_h creates a Merkle tree over all state hashes, i.e., $\mathsf{sTree}_\ell := \mathsf{MTree}(\{\mathsf{hash}^{(i)}_{(\ell,k)}\}_{k\in[R],i\in[n]})$, and broadcasts a signature on the root of this tree, i.e., $\langle \mathsf{MRoot}(\mathsf{sTree}_\ell)\rangle_h$.

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 ($\mathsf{initData}^{\mathsf{core}}_{(\ell,i)}$) 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. [6] in the context of verifiable computing. Later, the technique was used in [16, 20, 11].

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

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. [10, 8] and Wang et al. [21, 22, 24]. Input-independent PVC protocols with a public transcript can be obtained from semi-honest protocols using the input-independent compilers of Damgård et al.[9] and Faust et al. [13].

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 $\mathsf{state}^{(h)}_{(\ell,k)}$ with the outgoing messages $\{\mathsf{msg}^{(h,i)}_{(\ell,k)}\}_{i\in[n]\setminus\{h\}}$, then it actually sends the following to P_i:

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

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):

$$\{(\langle \mathsf{initData}^{\mathsf{core}}_{(i,\ell)}\rangle_i, \mathsf{initData}^{\mathsf{aux}}_{(i,\ell)})\}_{\ell\in O, i\in[n]}, \tag{1a}$$

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

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

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

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 $\mathsf{Blame}^{\mathsf{pp}}$ 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 $\mathsf{Blame}^{\mathsf{pp}}$

1. P_h runs $\mathsf{state}^{(i)}_{(\ell,0)} = \mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}_{(i,\ell)}, \mathsf{initData}^{\mathsf{aux}}_{(i,\ell)})$ for each $i \in [n], \ell \in O$. Let $\mathcal{B}$ be the set of all tuples $(\ell, 0, m, 0)$ such that $\mathsf{state}^{(m)}_{(\ell,0)} = \mathsf{bad}$. 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., $(\{\mathsf{msg}^{(i,j)}_{(\ell,k)}\}_{k\in[R],i\in[n],j\neq i}, \{\mathsf{state}^{(i)}_{(\ell,k)}\}_{k,i,j}) := \mathsf{emulate}^{\mathsf{full}}(\{\mathsf{state}^{(i)}_{(\ell,0)}\}_{i\in[n]})$.
3. Let $\mathcal{B}$ be the set of all tuples (ℓ, k, m, i) such that $H(\mathsf{msg}^{(m,i)}_{(\ell,k)}) \neq \mathsf{hash}^{(m,i)}_{(\ell,k)}$ or $H(\mathsf{state}^{(m)}_{(\ell,k)}) \neq \mathsf{hash}^{(m)}_{(\ell,k)}$ – where $\mathsf{hash}^{(m,i)}_{(\ell,k)}$ and $\mathsf{hash}^{(m)}_{(\ell,k)}$ 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 $\mathsf{cert} := \perp$. 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 $\perp$.

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

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

The punishment protocol. Each party P_i (for $i \in [n]$) checks if $\mathsf{cert} \neq \perp$. If this is the case, P_i sends cert to $\mathcal{J}^{\mathsf{pp}}$. 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 $\mathsf{Blame}^{\mathsf{pp}}$. The judge $\mathcal{J}^{\mathsf{pp}}$ is described in the following. The validation algorithms $\mathsf{wrongMsg}$ and $\mathsf{wrongState}$ and the algorithm $\mathsf{getIndex}$ can be found in the full version of this paper. We stress that the validation algorithms $\mathsf{wrongMsg}$ and $\mathsf{wrongState}$ don't need to recompute a whole protocol execution but only a single step. Therefore, $\mathcal{J}^{\mathsf{pp}}$ 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 $\mathcal{J}^{\mathsf{pp}}$

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 $\mathsf{cheaters}$ initially set to $\emptyset$. Prior to the execution of π^{pp}, $\mathcal{J}^{\mathsf{pp}}$ has received d coins from each party P_i.

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

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

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

6.1 Security

Theorem 1. *Let* $(\pi^{\mathsf{pp}}, \cdot, \cdot)$ *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* $(\mathsf{Generate}, \mathsf{Sign}, \mathsf{Verify})$ *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* $\mathsf{Blame}^{\mathsf{pp}}$*, protocol* $\mathsf{Punish}^{\mathsf{pp}}$ *and judge* $\mathcal{J}^{\mathsf{pp}}$ *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 $\mathsf{state}_{(\ell,k)}^{(h)}$ with the outgoing messages $\{\mathsf{msg}_{(\ell,k)}^{(h,i)}\}_{i\in[n]\setminus\{h\}}$, then it actually sends the following to P_i:

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

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.,

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

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

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

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

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

and the signed incoming message, i.e.,

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

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 $\mathsf{state}^{(i)}_{(\ell,0)} = \mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}_{(i,\ell)}, \mathsf{initData}^{\mathsf{aux}}_{(i,\ell)})$ for each $i \in [n], \ell \in O$. Let $\mathcal{B}$ be the set of all tuples $(\ell, 0, m, 0)$ such that $\mathsf{state}^{(m)}_{(\ell,0)} = \mathsf{bad}$. 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., $(\{\widetilde{\mathsf{msg}}^{(i,j)}_{(\ell,k)}\}_{k\in[R], i\in[n], j\neq i}, \{\mathsf{state}^{(i)}_{(\ell,k)}\}_{k,i,j}) := \mathsf{emulate}^{\mathsf{full}}(\{\mathsf{state}^{(i)}_{(\ell,0)}\}_{i\in[n]})$.
3. Let $\mathcal{B}$ be the set of all tuples $(\ell, k, m, h)$ such that $\mathsf{msg}^{(m,h)}_{(\ell,k)} \neq \widetilde{\mathsf{msg}}^{(m,h)}_{(\ell,k)}$ or $H(\mathsf{state}^{(m)}_{(\ell,k)}) \neq \mathsf{hash}^{(m)}_{(\ell,k)}$ – where $\mathsf{msg}^{(m,h)}_{(\ell,k)}$ and $\mathsf{hash}^{(m)}_{(\ell,k)}$ are taken from $\mathcal{M}$ or $\mathsf{sTree}_\ell$ respectively. In case of an invalid state, set $h = 0$.
4. Pick the tuple $(\ell, k, m, i)$ from $\mathcal{B}$ with the smallest $\ell$, $k$, $m$, $i$ in this sequence. If $k > 0$ set $\mathsf{msg}_{out} := \langle \mathsf{msg}^{(m,i)}_{(\ell,k)}\rangle_m$. Otherwise, set $\mathsf{msg}_{out} := \bot$.
5. Output partial certificate $(ids, \mathsf{msg}_{out})$.
```

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 $\mathsf{cert}' := \bot$.

Finally, each honest party executes the blame algorithm $\mathsf{Blame}^{\mathsf{sp}}$ 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 $\mathsf{cert}' = \bot$. The final certificate is generated by appending following data from the view to the certificate: $\{(\langle \mathsf{initData}^{\mathsf{core}}_{(i,\ell)}\rangle_j, \mathsf{initData}^{\mathsf{aux}}_{(i,\ell)})\}_{\ell\in O, i\in[n], j\in[n]}$ (cf. Eq 2a), $\{\mathsf{sTree}_\ell\}_{\ell\in[t]}$ (cf. Eq 2b), and $\{\langle \mathsf{MRoot}(\mathsf{sTree}_\ell)\rangle_i\}_{i\in[n],\ell\in[t]}$ (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 $\mathsf{cert}' \neq \bot$, the honest party outputs in addition to the certificate $\mathsf{corrupted}_m$.

To ease the specification of the punishment protocol in which parties derive further data from the certificates, we define an additional algorithm $\mathsf{mesHistory}$ that uses the messages obtained during the emulation $(\widetilde{\mathsf{msg}})$[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:

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

Additionally, if $\mathsf{cert}' \neq \bot$, parties compute the following:

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

The punishment protocol. The main difficulty of constructing a punishment protocol $\mathsf{Punish}^{\mathsf{sp}}$ for this scenario is that there is no publicly verifiable evidence about messages like a common transcript used in the previous transformations. 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 $\mathsf{Punish}^{\mathsf{sp}}$ into three phases. First, the judge determines the earliest accusation of misbehavior. To this end, if $\mathsf{cert} \neq \bot$ all parties start by sending tuple ids from cert to $\mathcal{J}^{\mathsf{sp}}$ 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.

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

If there has not been any accusation submitted in the first phase, $\mathcal{J}^{\mathsf{sp}}$ reimburses all parties. Otherwise, $\mathcal{J}^{\mathsf{sp}}$ 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 $\mathcal{J}^{\mathsf{sp}}$ 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, $\mathcal{J}^{\mathsf{sp}}$ 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:

<u>Protocol $\mathsf{Punish}^{\mathsf{sp}}$</u>

Phase 1: Determine earliest accusation

1. If $\mathsf{cert} \neq \bot$, P_h sends $ids := (\ell, k, m, i)$ taken from cert to $\mathcal{J}^{\mathsf{sp}}$ which stores (ℓ, k, m, i, h).
2. $\mathcal{J}^{\mathsf{sp}}$ 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, $\mathcal{J}^{\mathsf{sp}}$ sends d coins to each party at time T. Otherwise, $\mathcal{J}^{\mathsf{sp}}$ 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 $\mathcal{J}^{\mathsf{sp}}$ terminates by sending d coins to all parties but P_j.

Phase 2: First evidence

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

(d) If $i = 0$ and $\mathsf{wrongState}(\perp, \mathsf{state}_{in}, \mathsf{state}_{out}, \mathcal{M}_{in}, \mathsf{root}^{\mathsf{state}}_{(\ell)}, \mathsf{root}^{\mathsf{round}}_{k'}, \ell, k, m) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.

(e) If $i > 0$, $\mathsf{Verify}(\langle \mathsf{msg}^{(m,i)}_{(\ell,k)} \rangle_m) = \mathsf{false}$ or $\mathsf{wrongMsg}(\mathsf{state}_0, \mathsf{state}_{in}, H(\mathsf{msg}^{(m,i)}_{(\ell,k)}), \mathcal{M}_{in},, \mathsf{root}^{\mathsf{state}}_{(\ell)}, \mathsf{root}^{\mathsf{round}}_{k'}, \ell, m, k, i) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.

(f) $\mathcal{J}^{\mathsf{sp}}$ terminates by sending d coins to all parties but P_m.

Phase 3: Dispute the message tree

5. Parties P_b, P_m and $\mathcal{J}^{\mathsf{sp}}$ run bisection sub-protocol Π_{BS} on the top-level tree. P_b's input is the tree with root $\mathsf{root}^{\mathsf{msg}}$; P_m's the one with root $\widetilde{\mathsf{root}}^{\mathsf{msg}}$. $\mathcal{J}^{\mathsf{sp}}$ announces public inputs $\mathsf{root}^{\mathsf{msg}}$ and width of $\mathsf{root}^{\mathsf{msg}}$, $width := k'$. The output is the first round they disagree on k_2, the agreed hash $\mathsf{root}^{\mathsf{round}}_{k'_2}$ of leaf with index $k'_2 := k_2 - 1$ and the hash $\mathsf{root}^{\mathsf{round}}_{(b,k_2)}$ of leaf with index k_2 as claimed by P_m.
6. Parties P_m, P_b and $\mathcal{J}^{\mathsf{sp}}$ run bisection sub-protocol Π_{BS} on the low-level tree. Both, P_m and P_b take as input $\mathsf{mTree}^{\mathsf{round}}_{k_2}$ from their certificate. $\mathcal{J}^{\mathsf{sp}}$ announces public inputs $\mathsf{root}^{\mathsf{round}}_{(b,k_2)}$ and the width of the low level tree $width' n \times (n-1)$. 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 $i_2 = x \mod (n-1)$ if $m_2 > (x \mod (n-1))$ and $i_2 := (x \mod (n-1)) + 1$ otherwise.
7. Party P_b define variables as follows – variables that are not explicitly defined are set to $\perp$.

$$\begin{aligned} \text{(Always):} \quad \mathsf{initData}^2 &:= (\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m_2)} \rangle_m, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m_2)}) \\ \mathsf{root}^{\mathsf{state}} &:= \langle \mathsf{MRoot}(\mathsf{sTree}_\ell) \rangle_m \\ \text{(If } k_2 > 1\text{):} \quad \mathsf{state}^2_{in} &:= (\mathsf{state}^{(m_2)}_{(\ell,k'_2)}, \mathsf{MProof}(H(\mathsf{state}^{(m_2)}_{(\ell,k'_2)}), \mathsf{sTree}_\ell)) \\ \mathcal{M}^2_{in} &:= \{(\mathsf{msg}^{(j,m_2)}_{(\ell,k'_2)}, \mathsf{MProof}(H(\mathsf{msg}^{(j,m_2)}_{(\ell,k'_2)}), \mathsf{mTree}^{\mathsf{round}}_{k'_2}))\}_{j \in [n]} \end{aligned}$$

and sends $(\mathsf{initData}^2, \langle \mathsf{MRoot}(\mathsf{sTree}_\ell) \rangle_m, \mathsf{state}^2_{in}, \mathcal{M}^2_{in})$ to $\mathcal{J}^{\mathsf{sp}}$.
8. $\mathcal{J}^{\mathsf{sp}}$ parses $\mathsf{initData}^2$ to $(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m_2)} \rangle_m, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m_2)})$ and sets $\mathsf{state}^{(m_2)}_{(0)} := \mathsf{deriveInit}(\mathsf{initData}^{\mathsf{core}}_{(\ell,m_2)}, \mathsf{initData}^{\mathsf{aux}}_{(\ell,m_2)})$. If $\mathsf{Verify}(\langle \mathsf{root}^{\mathsf{state}}_{(\ell)} \rangle_m) = \mathsf{false}$, $\mathsf{Verify}(\langle \mathsf{initData}^{\mathsf{core}}_{(\ell,m_2)} \rangle_m) = \mathsf{false}$ or $\mathsf{state}^{(m_2)}_{(0)} \in \{\perp, \mathsf{bad}\}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
9. If $\mathsf{wrongMsg}(\mathsf{state}^{(m_2)}_{(0)}, \mathsf{state}^2_{in}, \mathsf{hash}_x, \mathcal{M}^2_{in}, \mathsf{root}^{\mathsf{state}}_{(\ell)}, \mathsf{root}^{\mathsf{round}}_{k'_2}, \ell, m_2, k_2, i_2) = \mathsf{false}$, $\mathcal{J}^{\mathsf{sp}}$ discards.
10. $\mathcal{J}^{\mathsf{sp}}$ terminates by sending d coins to all parties but P_m.

7.1 Security

Theorem 2. *Let* $(\pi_3^{\mathsf{pvc}}, \mathsf{Blame}^{\mathsf{pvc}}, \mathsf{Judge}^{\mathsf{pvc}})$ *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* $\mathsf{Blame}^{\mathsf{pvc}}$ *and* $\mathsf{Judge}^{\mathsf{pvc}}$*. Let* π_3 *be a protocol that is equal to* π_3^{pvc} *but does not generate a common transcript and instead of calling* $\mathsf{Blame}^{\mathsf{pvc}}$ *executes the blame procedure explained above (including execution of* $\mathsf{Evidence}$ *and* $\mathsf{Punish}^{\mathsf{sp}}$*). Further, let the signature scheme* $(\mathsf{Generate}, \mathsf{Sign}, \mathsf{Verify})$ *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* $\mathsf{Blame}^{\mathsf{sp}}$*, protocol* $\mathsf{Punish}^{\mathsf{sp}}$ *and judge* $\mathcal{J}^{\mathsf{sp}}$ *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 leave 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.

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

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 [12, 7]. 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 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 [25]. 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 [25] 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 [25] 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 [25] 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, [25] 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, [25] 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 [25] 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

1. Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *IEEE SP*, 2014.
2. Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In *ASIACRYPT*, 2012.
3. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, 2007.
4. Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, 1990.
5. Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO*, 2014.
6. Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In *CCS*, 2011.
7. CoinMarketCap. Ethereum (ETH) price. https://coinmarketcap.com/currencies/ethereum/, 2021.
8. 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.
9. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Black-box transformations from passive to covert security with public verifiability. In *CRYPTO*, 2020.
10. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.
11. Lisa Eckey, Sebastian Faust, and Benjamin Schlosser. Optiswap: Fast optimistic fair exchange. In *ASIA CCS*, 2020.
12. Etherscan. Ethereum Average Gas Price Chart. https://etherscan.io/chart/gasprice, 2021.
13. Sebastian Faust, Carmit Hazay, David Kretzler, and Benjamin Schlosser. Generic compiler for publicly verifiable covert multi-party computation. In *EUROCRYPT*, 2021.
14. 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.
15. Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wen-jie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. In *EUROCRYPT*, 2019.
16. Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *USENIX Security*, 2018.
17. Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. In *ASIACRYPT*, 2015.
18. Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In *CCS*, 2014.
19. Peter Scholl, Mark Simkin, and Luisa Siniscalchi. Multiparty computation with covert security and public verifiability. *IACR Cryptol. ePrint Arch.*, 2021.

20. Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *CoRR*, abs/1908.04756, 2019.
21. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, 2017.
22. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.
23. Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.
24. Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In *CCS*, 2020.
25. Ruiyu Zhu, Changchang Ding, and Yan Huang. Efficient publicly verifiable 2pc over a blockchain with applications to financially-secure computations. In *CCS*, 2019.

E. Non-Interactive Threshold BBS+ from Pseudorandom Correlations

In this chapter, we present the following work with minor changes.

[93] S. Faust, C. Hazay, D. Kretzler, L. Rometsch, and B. Schlosser. *Non-Interactive Threshold BBS+ From Pseudorandom Correlations.* Cryptology ePrint Archive, Paper 2023/1076. `https://eprint.iacr.org/2023/1076`. 2023. **Part of this thesis**.

Non-Interactive Threshold BBS+ From Pseudorandom Correlations

Sebastian Faust[1], Carmit Hazay[2], David Kretzler[1], Leandro Rometsch[1], and Benjamin Schlosser[1]

[1] Technical University of Darmstadt, Germany
{first.last}@tu-darmstadt.de
[2] Bar-Ilan University, Israel
carmit.hazay@biu.ac.il

Abstract. The BBS+ signature scheme is one of the most prominent solutions for realizing anonymous credentials. Its prominence is due to properties like selective disclosure and efficient protocols for creating and showing possession of credentials. Traditionally, a single credential issuer produces BBS+ signatures, which poses significant risks due to a single point of failure.
In this work, we address this threat via a novel t-out-of-n threshold BBS+ protocol. Our protocol supports an arbitrary security threshold $t \leq n$ and works in the so-called preprocessing setting. In this setting, we achieve non-interactive signing in the online phase and sublinear communication complexity in the number of signatures in the offline phase, which, as we show in this work, are important features from a practical point of view. As it stands today, none of the widely studied signature schemes, such as threshold ECDSA and threshold Schnorr, achieve both properties simultaneously. To this end, we design specifically tailored presignatures that can be directly computed from pseudorandom correlations and allow servers to create signature shares without additional cross-server communication. Both our offline and online protocols are actively secure in the Universal Composability model. Finally, we evaluate the concrete efficiency of our protocol, including an implementation of the online phase and the expansion algorithm of the pseudorandom correlation generator (PCG) used during the offline phase. The online protocol without network latency takes less than $15ms$ for $t \leq 30$ and credentials sizes up to 10. Further, our results indicate that the influence of t on the online signing is insignificant, $< 6\%$ for $t \leq 30$, and the overhead of the thresholdization occurs almost exclusively in the offline phase. Our implementation of the PCG expansion is the first considering correlations between more than 3 parties and shows that even for a committee size of 10 servers, each server can expand a correlation of up to 2^{16} presignatures in about 600 ms per presignature.

Keywords: Threshold Signature · BBS+ · Pseudorandom Correlation Functions · Pseudorandom Correlation Generators

1 Introduction

Anonymous credentials schemes, as introduced by Chaum in 1985 [32] and subsequently refined by a line of work [33, 56, 26, 27, 22, 24, 25, 10, 73], allow an issuing party to create credentials for users, which then can prove individual attributes about themselves without revealing their identities. The essential properties these schemes satisfy are *unlinkability*, ensuring that verifiers cannot link two disclosures of credentials of the same identity, and *selective disclosure*, allowing parties to decide which individual attributes of their credentials to disclose. The former makes anonymous credentials a useful privacy tool on the web, allowing clients to authenticate themselves for access to web-based services while preventing service providers from gathering information about the client's usage patterns. The latter makes anonymous credentials an essential building block for self-sovereign identity frameworks, as it enables clients to not only take responsibility for storing their credentials but also to filter the disclosure of their credentials.

The BBS+ signature scheme [5, 23] named after the group signature scheme of Boneh, Boyen, and Shacham [11] is one of the most prominent solutions for realizing anonymous credential schemes. Abstractly speaking, a BBS+ signature over a set of attributes constitutes credentials, and the holder of such a credential can prove possession of individual attributes using efficient zero-knowledge protocols. BBS+ signatures are particularly suited for anonymous credentials because of their appealing features, including the ability to sign an array of attributes while keeping the signature size constant, efficient protocols for blind signing, and efficient zero-knowledge proofs for selective disclosure of signed attributes (without having to reveal the signature). The importance of BBS+ is illustrated by the renewed attention in the research community [67, 42], several industrial implementations [68, 57, 58], ongoing standardization efforts by the W3C Verifiable Credentials Group and IETF [9, 55], and adaption in further real-world applications [5, 34, 20, 21, 23].

In traditional credential systems, the credential issuer who is in possession of the signing key constitutes a single point of failure. A powerful and widely adapted tool mitigating such a single point of failure is to distribute the cryptographic task (e.g., [53, 46, 54, 43, 65, 30, 29, 49, 50, 2, 31, 35] and many more) via so-called *threshold cryptography*. Here, the cryptographic key is shared among a set of servers such that any subset of t servers can produce a signature, while the underlying signature scheme remains secure even if up to $t - 1$ servers are corrupted. The thresholdization of digital signature schemes comes with significant overhead in computation, communication, and round complexity. This is particularly the case for randomized signature schemes, where a random secret nonce has to be generated among a set of servers. In the signing protocol, this nonce is then used together with the shared key to produce the signature. Concretely, for BBS+ signing, we require a distributed protocol to compute the exponentiation of the inverse of the secret key added to the random nonce securely.

The straightforward approach to compute the inverse is based on the inversion protocol by Bar-Ilan and Beaver [7] and requires server interaction. In order to strengthen the protection against failure and corruption, we assess it as likely for servers to be located in different jurisdictional and geographic regions. In such a setting, any additional communication round involves a significant performance overhead. Therefore, an ideal threshold BBS+ scheme has a non-interactive signing phase that enables servers to respond to signature requests without any cross-server interaction.

A popular approach in secure distributed computation to cope with the high complexities of protocols is to split the computation into an input-independent offline and input-dependent online phase [39, 59, 69, 70]. The offline phase provides precomputation material, which in the setting of a digital signature scheme is called presignatures [44]. These presignatures are produced during idle times of the system and facilitate an efficient online phase. In recent years, Boyle et al. [14, 17, 18] put forth a novel concept to generate precomputation material called *pseudorandom correlation-based precomputation (PCP)*. The main advantage of this concept is the generation of precomputation material in sublinear communication complexity in the amount of generated precomputation material. Recently, this technique also attracted interest for use in threshold signature protocols [2, 51]. In PCP, precomputed values are generated by a pseudorandom correlation generator (PCG) or a pseudorandom correlation function (PCF). These primitives include an interactive setup phase where short keys are generated and distributed. Then, in the evaluation phase, every party locally evaluates on its key and a common input. The outputs look pseudorandom but still satisfy some correlation, e.g., oblivious linear evaluation (OLE), oblivious transfer (OT), or multiplication triples.

1.1 Contribution

We propose a novel t-out-of-n threshold BBS+ signature scheme in the offline-online model with an arbitrary security threshold $t \leq n$. The centerpiece of our protocol is the design of specifically tailored presignatures that can be directly instantiated from PCG or PCF evaluations and can be used by servers to create signature shares without any additional cross-server communication. This way, our scheme simultaneously provides a non-interactive online signing phase and an offline phase with sublinear communication complexity in the number of signatures. Thus, our protocol is the first threshold BBS+ signature scheme with non-interactive signing. Even for the widely studied signature schemes ECDSA and Schnorr, no threshold protocol exists that achieves both features simultaneously. Moreover, we are the first to present a PCG/PCF-based protocol that supports t-out-of-n threshold, while previous protocols support only n-out-of-n. We formally analyze the static security of all our protocols in the Universal Composability framework under active corruption.

We present two instantiations of the offline phase, one based on PCGs and one based on PCFs. Conceptually, PCFs are better suited than PCGs for preprocessing signatures as PCFs allow servers to compute presignatures only when needed. In contrast, PCGs require the generation of a large batch of presignatures at once that need to be stored on the server side. Nevertheless, existing PCG constructions provide better efficiency than PCF constructions. Therefore, we present protocols for both primitives.

Unlike prior work using silent preprocessing in the context of threshold signatures [2], we use the PCG and PCF primitive in a black-box way, allowing for a modular treatment. In this process, we identify several issues in using the primitives in a black-box way, extend the definitional frameworks accordingly, and prove the security of existing constructions under the adapted properties.

On a practical level, we provide an extensive evaluation of our protocol, including an implementation and experimental evaluation of the online phase and the seed expansion of the PCG-based offline phase. Since state-of-the-art PCF constructions lack concrete efficiency, we focus our evaluation on the PCG-based preprocessing. Given preprocessed presignatures, the total runtime of the online signing protocol is below 13.595 ms plus one round trip time of the slowest client-server connection for $t \leq 30$ signers and message arrays of size $k \leq 10$. Our benchmarks show that the influence of the number of signers on the runtime of the online protocol is minimal; increasing the number of signers from 2 to 30 increases the runtime by just $1.14\% - 5.52\%$ (for message array sizes between 2 and 50). Further, our results show that the cost of thresholdization occurs almost exclusively in the offline phase; a threshold signature on a single message array takes 7.536 ms in our protocol, while a non-threshold signature, including verification of the received signature, takes 7.248 ms; ignoring network delays which are the same in both settings. Our implementation of the PCG seed expansion is the first to consider more than 3 parties. In our benchmarks, we extend batches of up to 2^{16} presignatures for $2 \leq n \leq 10$ parties in both the n-out-of-n and the t-out-of-n setting. Even when considering the t-out-of-10 setting and batches of 2^{16} presignatures, the computation time per signature is roughly 600 ms. Our results show that the computation cost increases linearly with the number of parties and superlinear with the size of the presignature batches. However, our complexity analysis shows that the PCG key size and the communication of a distributed key generation protocol grow sublinear, leading to a trade-off between communication and computation complexity.
We summarize our contribution as follows:

- We propose the first threshold BBS+ scheme with a non-interactive online signing phase.
- Our scheme simultaneously achieves non-interactive online signing and sublinear communication in the offline phase. This combination is not achieved by the widely studied threshold protocols for ECDSA and Schnorr.
- We extend the definitional framework of PCGs and PCFs by introducing the notion of *(strong) reusability* for both primitives.
- We specify two instantiations for the offline phase, one based on PCGs and one based on PCFs.
- We prove the static security of our protocols in the Universal Composability framework with active corruption.
- We provide an evaluation of the whole protocol with the PCG-based precomputation.
- We provide an implementation and evaluation of the online phase and the PCG-based offline phase's seed expansion.

For the sake of presentation, we focus the main body on PCGs and present the definition of reusable PCFs and the PCF-based offline phase in the Appendix.

1.2 Technical Overview

BBS+ signatures. Let $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_T$ be groups of prime order p with generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ and let map $\mathsf{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be a bilinear paring. A BBS+ signature on a message array $\{m_\ell\}_{\ell \in [k]}$ is a tuple (A, e, s) with $A = (g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$ for random nonces $e, s \in_R \mathbb{Z}_p$, secret key $x \in \mathbb{Z}_p$ and a set of random elements $\{h_\ell\}_{\ell \in [0..k]}$ in $\mathbb{G}_1$. To verify under public key g_2^x, check if $\mathsf{e}(A, g_2^x \cdot g_2^e) = \mathsf{e}(g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell}, g_2)$ (see Appendix A for a formal description).

Distributed inverse calculation. The main difficulty in thresholdizing the BBS+ signature algorithm comes from the signing operation requiring the computation of the inverse of $x + s$ without leaking x. This highly non-linear operation is expensive to be computed in a distributed way. Similar challenges are known from other signature schemes relying on exponentiation (or a scalar multiplication in additive notion) of the inverse of secret values, e.g., ECDSA [6, 29, 2, 72, 13]. The typical approach (cf. [7]) to compute $M^{\frac{1}{y}}$ for a group element M and a secret shared y is to separately open $B = M^a$ and $\delta = a \cdot y$ for a freshly shared random a. The desired result can be reconstructed by computing $M^{\frac{1}{y}} = B^{\frac{1}{\delta}}$.

Since δ is the product of two secret shared values, it still is a non-linear operation requiring interaction between the parties. Nevertheless, as δ is independent of the actual message, several such values can be precomputed in an *offline* phase. As explained next, a similar, yet more involved, approach can be applied to the BBS+ protocol, allowing an efficient, non-interactive online signing based on correlated precomputation material.

The threshold BBS+ online protocol. We describe a simplified, n-out-of-n version of our threshold BBS+ protocol. Assume a BBS+ secret key x, elements $\{h_\ell\}_{\ell \in [0..k]}$ in $\mathbb{G}_1$, a random blinding factor $a \in \mathbb{Z}_p$ and n servers, each having access to a preprocessed tuple $(a_i, e_i, s_i, \delta_i, \alpha_i) \in \mathbb{Z}_p^5$, in the following called presignatures, such that

$$\begin{aligned} \delta = \sum_{i \in [n]} \delta_i = a(x+e), \qquad \sum_{i \in [n]} \alpha_i = as \\ \text{for } a = \sum_{i \in [n]} a_i, \qquad e = \sum_{i \in [n]} e_i, \qquad s = \sum_{i \in [n]} s_i. \end{aligned} \tag{1}$$

To sign a message array $\{m_\ell\}_{\ell\in[k]}$, each server computes $A_i := (g_1 \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell})^{a_i} \cdot h_0^{\alpha_i}$ and outputs a partial signature $\sigma_i := (A_i, \delta_i, e_i, s_i)$. This allows the receiver of the partial signatures to reconstruct δ, e and s and compute

$$A = (\prod_{i\in[n]} A_i)^{\frac{1}{\delta}} = ((g_1 \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell})^a \cdot h_0^{as})^{\frac{1}{a(x+e)}}$$

such that the tuple (A, e, s) constitutes a valid BBS+ signature. Each signature requires a new preprocessed tuple to prevent straightforward forgeries.

The specialized layout of our presignatures allows us to realize a non-interactive signing procedure. In contrast, using plain multiplication triples, as often done in multi-party computation protocols [8, 39], would require one additional round of communication. Further, our online protocol provides active security at a low cost. This is achieved by verifying the received signatures and works since the presignatures are created securely.

The preprocessing protocol. An appealing choice for instantiating the preprocessing protocol is to use pseudorandom correlation generators (PCG) or functions (PCF), as they enable the efficient generation of correlated random tuples. More precisely, PCGs and PCFs allow two parties to expand short seeds to fresh correlated random tuples locally. While the distributed generation of the seeds requires more involved protocols and typically relies on general-purpose multi-party computation, the seed size and the communication complexity of the generating protocols are sublinear in the size of the expanded correlated tuples [14, 17].

The correlated pseudorandom presignatures required by our online signing procedure are specifically tailored to the BBS+ setting (cf. (1)). For these specific presignatures, there exist no tailored PCG or PCF constructions. Instead, we show how to obtain these presignatures from simple correlations. Specifically, we leverage *oblivious linear evaluation* (OLE) and *vector oblivious linear evaluation* (VOLE) correlations. For both of these correlations, there exist PCG and PCF constructions [14, 17, 18, 19, 36, 60, 15]. An OLE tuple is a two-party correlation, in which party P_1 gets random values (a, u) and party P_2 gets random values (s, v) such that $a \cdot s = u + v$. A VOLE tuple provides the same correlation but fixes s over all tuples computed by the particular PCG or PCF instance. In these tuples, we call a and s the *input* value of party P_1 and P_2. Further, the PCGs/PCFs used by our protocol provide a so-called *reusability* feature, allowing parties to fix the *input* values over several PCG/PCF instances. This feature enables parties to turn two-party into multi-party correlations as shown by [19, 2, 3]. It is achieved by extending the definitions with the ability of both parties to provide parameters to the key generation.

For computing the product of two secret shared values, a and s, the parties use OLE correlations. Let $\alpha = \sum_{i\in[n]} a_i \cdot \sum_{j\in[n]} s_j$, where a_i and s_i are known to party P_i. Only $a_i s_i$ can be locally computed by P_i. For all cross terms $a_i s_j$ for $i \neq j$, the parties use OLE correlations to get an additive share of that cross term, i.e., $a_i s_j = u_{i,j} + v_{i,j}$. By adding $a_i s_i$ to the sum of all additive shares $u_{i,j}$ and $v_{j,i}$, party P_i obtains an additive share of α. Note that the a_i value must be the same for all cross terms, so we require the OLE PCG/PCF to provide the reusability feature. This allows party P_i to use the same input value a_i in all OLE correlations for the cross terms $a_i s_j$ with $j \neq i$.

Using PCGs/PCFs in a black-box way. Pseudorandom correlation generators (PCGs) and pseudorandom correlation functions (PCFs) are introduced in [14] and [18]. Concrete constructions of both primitives for simple correlations, such as VOLE, are presented in a line of work including [14, 17, 16, 19, 18, 36, 60]. In our work, we aim to deal with PCGs/PCFs in a black-box way such that we can instantiate our protocols with arbitrary constructions as long as they fulfill our requirements. These requirements include supporting VOLE and OLE correlations, the active security setting, and the opportunity to reuse inputs, as emphasized above. A first step towards black-box usage of PCGs was taken by [17]. This work defines an ideal functionality for correlated randomness, which they show can be instantiated by PCGs. However, the definition does not support reusing inputs to PCGs.

[17] and [18] lay the foundation of the reusability property for PCGs and PCFs. However, their definitions consider passive security only and are unsuitable for black-box usage. Therefore, we introduce new notions called *reusable PCG* and *reusable PCF*, which capture the active security setting and permit black-box use.

Identical to prior definitions of PCGs and PCFs, our primitives consist of a key generation Gen and an expansion algorithm Expand or evaluation algorithm Eval. The reusability feature allows both parties to specify an input to the key generation, which is used to derive the correlation tuples. Additionally, our reusable primitives must satisfy four properties. Three of these properties are stated by [17] and [18], two of which we slightly modified. Our new insight is the requirement of the *key indistinguishability* property, which we specifically introduce to cover malicious parties. The key indistinguishability property states that the adversary cannot learn information about the honest party's input to the key generation, even if the corrupted party chooses its input arbitrarily. This property makes our notion suitable for the active security setting.

We present reusable PCG constructions for VOLE and OLE correlations and prove that the VOLE PCF construction by Boyle et al. [18] fulfills our new definition. Additionally, we present an extension of this construction for OLE correlations.

The t-out-of-n setting. So far, we discussed a setting where n-out-of-n servers must contribute to the signature creation. However, in many use cases, we need to support the more flexible t-out-of-n setting with $t \leq n$. In this setting, the secret key material is distributed to n servers, but only t must contribute to the signing protocol. A threshold $t \leq n$ improves the flexibility and robustness of the signing process, as not all servers must be online.

The typical approach in the t-out-of-n setting is to share the secret key material using Shamir's secret sharing [64] instead of an additive sharing as done above. While additive shares are reconstructed by summation, Shamir-style shares must be aggregated using Lagrange interpolation, either on the client or server side. In this work, we reconstruct on the server side due to technical details of our precomputation protocols. Note that prior threshold signature schemes leveraging PCF/PCGs (e.g., [2, 51]) achieve only n-out-of-n, in contrast to a flexible t-out-of-n setting.

On a technical level, the challenge for client-side reconstruction is due to (V)OLE correlations providing us with two-party additive sharings of multiplications, e.g., $u_{i,j} + v_{i,j} = a_i s_j$. For a product of two additively shared values $a \cdot s$, we can rewrite the product as $\sum_{i\in[n]} a_i \cdot \sum_{i\in[n]} s_i = \sum_{i\in[n]} \sum_{j\in[n]} a_i s_j = \sum_{i\in[n]} \sum_{j\in[n]} u_{i,j} + v_{i,j}$. Here, $u_{i,j}$ and $v_{i,j}$ can be interpreted as additive shares of the product. These additive shares are sufficient for the n-out-of-n setting. However, it is unclear how (V)OLE outputs can be transformed to Shamir sharing of $a \cdot s$ required for t-out-of-n with client reconstruction.

We, therefore, incorporate a share conversion mechanism from Shamir-style shared key material into additively shared presignatures on the server side. Our mechanism consists of the servers applying the corresponding Lagrange interpolation directly to the outputs of the VOLE correlation. More precisely, as described above, each party P_i gets additive shares of the cross terms $a_i x_j$ and $a_j x_i$ for every other party P_j. Here, x_ℓ denotes the Shamir-style share of the secret key belonging to party P_ℓ. Let $c_{i,j}$ be the additive share of $a_i x_j$, then party P_i multiplies the required Lagrange coefficient $L_{j,\mathcal{T}}$ to this share and $L_{i,\mathcal{T}}$ to $c_{j,i}$, where $\mathcal{T}$ is the set of t

signers. The client provides the set of servers as part of the signing request to enable the servers to compute the interpolation.

1.3 Related Work

Most related to our work are the works by Gennaro et al. [47] and Doerner et al. [42], proposing threshold protocols for the BBS+ signing algorithm. While [47] focuses on a group signature scheme with threshold issuance based on the BBS signatures, their techniques can be directly applied to BBS+. [42] presents a threshold anonymous credential scheme based on BBS+. Both schemes compute the inverse using classical techniques of Bar-Ilan and Beaver [7]. Moreover, they realize the multiplication of two secret shared values by multiplying each pair of shares. While [47] uses a three-round multiplication protocol based on an additively homomorphic encryption scheme, [42] integrates a two-round OT-based multiplier. Although the OT-based multiplier requires a one-time setup, both schemes do not use precomputed values per signing request. This is in contrast to our scheme but at the cost of requiring several rounds of communication during the signing. Parts of their protocols are independent of the message that will be signed; thus, in principle, these steps can be moved to a presigning phase. In this case, the signing phase is non-interactive, but on the downside, the communication complexity of the presigning phase has linear complexity. In contrast, our protocol achieves both a non-interactive online phase and an offline phase with sublinear complexity. In addition, both works [47, 42] consider a security model tailored to the BBS+ signature scheme while we show security with respect to a more generic threshold signature ideal functionality.

In the non-threshold setting, Tessaro and Zhu [67] show that short BBS+ signatures, where the signature consists only of A and e, are also secure under the q-SDH assumption. Their results suggest removing s to reduce the signature size to one group element and a scalar. Like prior proofs of BBS+, their security proof in the standard model incurs a multiplicative loss. However, they present a tight proof in the Algebraic Group Model [45]. We discuss the impact of their work on our evaluation in Appendix N.

Another anonymous credential scheme with threshold issuance, called Coconut, is proposed by Sonnino et al. [66] and the follow-up work by Rial and Piotrowska [63]. Their scheme is based on the Pointcheval-Sanders (PS) signature scheme, which allows them to have a non-interactive issuance phase without coordination or precomputation. We emphasize that the PS signature scheme is less popular than BBS+ and not subject to standardization efforts. The security of PS and Coconut is based on a modified variant of the LRSW assumption introduced in [62]. This assumption is interactive in contrast to the q-Strong Diffie-Hellman assumption on which the security of BBS+ is based. While PS and Coconut also support multi-attribute credentials, the secret and public key size increases linearly in the number of attributes. In BBS+, the key size is constant. Further, PS and, therefore, the Coconut scheme relies on Type-3 pairings, while our scheme can be instantiated with any pairing type. The security of Coconut was not shown under concurrent composition while our scheme is analyzed in the Universal Composability framework.

Like our work, [2] and [51] leverage pseudorandom correlations for threshold signatures. [2] presents an ECDSA scheme, while [51] focuses on Schnorr signatures. [2] constructs a tailored PCG generating ECDSA- presignatures while our scheme uses existing VOLE and OLE PCGs/PCFs in a black-box way and combines the OLE and VOLE correlations to BBS+ presignatures. Further, in contrast to our work, [2] presents an n-out-of-n protocol without a flexible threshold. [51] introduces the new notion of a discrete log PCF and constructs a two-party protocol based on this primitive. In contrast to our work, [51] captures only the 2-out-of-2 setting. Both schemes [2, 51] require additional per-presignature communication. Depending on the phase this communication is assigned to, the schemes either have linear communication in the offline phase or require two rounds of communication in the online phase.

2 Preliminaries

Throughout this work, we denote the security parameter by $\lambda \in \mathbb{N}$, the set $\{1, \ldots, k\}$ as $[k]$, the set $\{0, 1, \ldots, k\}$ as $[0..k]$, the number of parties by n and a specific party by P_i. The set of indices of corrupted parties is denoted by $\mathcal{C} \subsetneq [n]$ and honest parties are denoted by $\mathcal{H} = [n] \setminus \mathcal{C}$. We denote vectors of elements via bold letters, e.g., $\mathbf{a}$, and the i-th element of a vector $\mathbf{a}$ by $\mathbf{a}[i]$.

We model our protocol in the Universal Composability (UC) framework by Canetti [28]. We refer to Appendix B for a brief introduction to UC. In our constructions, we denote by $\mathcal{Z}$ the UC environment and use sid and ssid to denote session and subsession identifier. We model a malicious adversary corrupting up to $t - 1$ parties. We consider static corruption and a rushing adversary. Our protocols are in the synchronous communication model.

We make use of a bilinear mapping following the definition of [12, 11]. A bilinear mapping is described by three cyclic groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ of prime order p, generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$, and a pairing $\mathsf{e} : \mathbb{G}_1 \times \mathbb{G} \to \mathbb{G}_T$. We call e a bilinear map iff it can be computed efficiently, $\mathsf{e}(u^a, v^b) = \mathsf{e}(u, v)^{ab}$ for all $(u, v, a, b) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{Z}_p \times \mathbb{Z}_p$, and $\mathsf{e}(g_1, g_2) \neq 1$ for all generators g_1 and g_2. We refer to [12] for a more formal specification.

3 Reusable Pseudorandom Correlation Generators

In this section we introduce our definition of reusable PCGs, extending the definition of programmable PCGs from [17] and [19]. We argue why existing constructions for PCGs satisfy our new definition in Appendix D. The extended definitional framework for PCFs and the PCF-based instantiation of the precomputation are stated in Appendix E and G.

In a nutshell, pseudorandom correlation generators allow two parties to generate a large amount of correlated randomness from short seeds. They are useful in many two- and multi-party protocols in the offline-online-model [39, 59, 69, 70]. Examples for frequently used correlations are oblivious linear evaluations, oblivious transfer and multiplication triples.

Our modifications and extensions of the definitions of [16] and [19] reflect the challenges we faced when using PCGs as black-box primitives in our threshold BBS+ protocol. We present our definition and highlight these challenges and changes in the following. We note that Boyle et al. [17] presents an ideal functionality for corruptible, correlated randomness which can be instantiated by PCGs. While this simulation-based notion allows to abstract from concrete PCG constructions, their ideal functionality does not cover the reusability feature required in our setting. Therefore, we present a suitable game-based definition.

3.1 Definition

As mentioned above, a PCF/PCG realizes a target correlation $\mathcal{Y}$. For some correlations, like VOLE, parts of the correlation outputs are fixed over all outputs. In the example of VOLE, where the correlation is $v = as + u$ over some ring R, the s value is fixed for all tuples.

Additionally, in a multi-party setting, we like PCG/PCF constructions that allow parties to obtain the same values for parts of the correlation output in multiple instances. Concretely, assume party P_i evaluates one VOLE PCG/PCF instance with party P_j and one with party P_k. P_i evaluates the PCG/PCF to $(a_{i,j}, u_{i,j})$ for the first instance and $(a_{i,k}, u_{i,k})$ for the second instance. Here, we want to give party P_i the opportunity to get $a_{i,j} = a_{i,k}$ when applied on the same input. This property is necessary to construct multi-party correlations from two-party PCG/PCF instances.

To formally model the abovementioned properties, we define a *target correlation* as a tuple of probabilistic algorithms $(\mathsf{Setup}, \mathcal{Y})$, where Setup takes two inputs and creates a master key mk. These inputs enable fixing parts of the correlation, e.g., the fixed value s in VOLE correlations, and obtaining the same values over multiple instances, e.g., by fixing to the same values s in multiple VOLE correlations. Algorithm $\mathcal{Y}$ uses the master key to sample correlation outputs.

Finally, we follow [16, 19] and require a target correlation to be reverse-sampleable to facilitate a suitable definition of PCGs. In contrast to [17, 19], our definition of a target correlation explicitly considers the reusability of values over multiple invocations.

Definition 1 (Reverse-sampleable target correlation with setup). *Let $\ell_0(\lambda), \ell_1(\lambda) \leq \mathsf{poly}(\lambda)$ be output length functions. Let $(\mathsf{Setup}, \mathcal{Y})$ be a tuple of probabilistic algorithms, such that Setup on input 1^λ and two parameters ρ_0, ρ_1 returns a master key mk; algorithm $\mathcal{Y}$ on input 1^λ and mk returns a pair of outputs $(y_0^{(i)}, y_1^{(i)}) \in \{0,1\}^{\ell_0(\lambda)} \times \{0,1\}^{\ell_1(\lambda)}$.*

We say that the tuple $(\mathsf{Setup}, \mathcal{Y})$ defines a reverse-sampleable target correlation with setup *if there exists a probabilistic polynomial time algorithm $\mathsf{RSample}$ that takes as input $1^\lambda, \mathsf{mk}, \sigma \in \{0,1\}, y_\sigma^{(i)} \in \{0,1\}^{\ell_\sigma(\lambda)}$ and outputs $y_{1-\sigma}^{(i)} \in \{0,1\}^{\ell_{1-\sigma}(\lambda)}$, such that for all $\sigma \in \{0,1\}$, for all $\mathsf{mk}, \mathsf{mk}'$ in the range of Setup for arbitrary but fixed input ρ_σ the following distributions are statistically close:*

$$\{(y_0, y_1) | (y_0, y_1) \xleftarrow{\$} \mathcal{Y}(1^\lambda, \mathsf{mk})\}$$
$$\{(y_0, y_1) | (y'_0, y'_1) \xleftarrow{\$} \mathcal{Y}(1^\lambda, \mathsf{mk}'),$$
$$y_\sigma \leftarrow y'_\sigma, y_{1-\sigma} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{mk}, \sigma, y_\sigma)\}.$$

Given the definition of a reverse-sampleable correlation with setup, we define our primitive called *reusable PCG (rPCG)*.

The security properties in the original notion of programmable PCGs assumes randomly selected seeds that are inserted into the key generation. This reflects a passive or semi-honest setting in which the adversary cannot deviate from the protocol description such that the seeds are indeed random. We are interested in the active security setting, where an adversary can insert arbitrary seeds into the key generation. Therefore, we propose the notion of *reusable pseudorandom correlation generators.*

Definition 2 (Reusable pseudorandom correlation generator (rPCG)). *Let $(\mathsf{Setup}, \mathcal{Y})$ be a reverse-sampleable correlation with setup which has output length functions $\ell_0(\lambda), \ell_1(\lambda)$, let $\lambda \leq \eta(\lambda) \leq \mathsf{poly}(\lambda)$ be the sample size function. Let $(\mathsf{PCG.Gen_p}, \mathsf{PCG.Expand})$ be a pair of algorithms with the following syntax:*

- *$\mathsf{PCG.Gen_p}(1^\lambda, \rho_0, \rho_1)$ is a probabilistic polynomial-time algorithm that on input the security parameter 1^λ and reusable inputs ρ_0, ρ_1 outputs a pair of keys $(\mathsf{k}_0, \mathsf{k}_1)$.*
- *$\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)$ is a deterministic polynomial-time algorithm that on input $\sigma \in \{0,1\}$ and key k_σ outputs $\mathbf{y}_\sigma \in \{0,1\}^{\ell_\sigma(\lambda) \times \eta(\lambda)}$, i.e. an array of size $\eta(\lambda)$ with elements being bit-strings of length $\ell_\sigma(\lambda)$.*

We say $(\mathsf{PCG.Gen_p}, \mathsf{PCG.Expand})$ is a reusable pseudorandom correlation generator (rPCG) for $(\mathsf{Setup}, \mathcal{Y})$, if the following conditions hold:

– ***Programmability.*** *There exist public efficiently computable functions* ϕ_0, ϕ_1*, such that for all* $\rho_0, \rho_1 \in \{0,1\}^*$

$$\Pr\left[\begin{array}{l}(\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen_p}(1^\lambda, \rho_0, \rho_1) \\ (\mathbf{x}_0, \mathbf{z}_0) \leftarrow \mathsf{PCG.Expand}(0, \mathsf{k}_0), \\ (\mathbf{x}_1, \mathbf{z}_1) \leftarrow \mathsf{PCG.Expand}(1, \mathsf{k}_1)\end{array} : \begin{array}{c}\mathbf{x}_0 = \phi_0(\rho_0) \\ \mathbf{x}_1 = \phi_1(\rho_1)\end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

– ***Pseudorandom*** $\mathcal{Y}$***-correlated outputs.*** *For every non-uniform adversary* $\mathcal{A}$ *of size* $\mathsf{poly}(\lambda)$ *it holds that*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{pr\text{-}g}}_{\mathcal{A}}(\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$$

for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{pr\text{-}g}}_{\mathcal{A}}(\lambda)$ *is as defined in Figure 1.*

– ***Security.*** *For each* $\sigma \in \{0,1\}$ *and non-uniform adversary* $\mathcal{A}$ *of size* $\mathsf{poly}(\lambda)$*, it holds that*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{sec\text{-}g}}_{\mathcal{A},\sigma}(\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$$

for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{sec\text{-}g}}_{\mathcal{A},\sigma}(\lambda)$ *is as defined in Figure 1.*

– ***Key indistinguishability.*** *For any* $\sigma \in \{0,1\}$ *and non-uniform adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$*, it holds*

$$\Pr[\mathsf{Exp}^{\mathsf{key\text{-}ind\text{-}g}}_{\mathcal{A},\sigma}(\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{key\text{-}ind\text{-}g}}_{\mathcal{A},\sigma}$ *is as defined in Figure 1.*

3.2 Correlations

Our OLE correlation of size N over a finite field $\mathbb{F}_p$ is given by $\mathbf{z}_1 = \mathbf{x}_0 \cdot \mathbf{x}_1 + \mathbf{z}_0$, where $\mathbf{x}_0, \mathbf{x}_1, \mathbf{z}_0, \mathbf{z}_1 \in \mathbb{F}_p^N$. Moreover, we require $\mathbf{x}_0$ and $\mathbf{x}_1$ being computed by a pseudorandom generator (PRG). Formally, we define the reverse-sampleable target correlation with setup $(\mathsf{Setup_{OLE}}, \mathcal{Y}_{\mathsf{OLE}})$ of size N over a field $\mathbb{F}_p$ as

$$\begin{gathered}\mathsf{mk} = (\rho_0, \rho_1) \leftarrow \mathsf{Setup_{OLE}}(1^\lambda, \rho_0, \rho_1)\,, \\ ((F_0(\rho_0), \mathbf{z}_0), (F_1(\rho_1), \mathbf{z}_1)) \leftarrow \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, \mathsf{mk}) \quad \text{such that} \\ \mathbf{z}_0 \xleftarrow{\$} \mathbb{F}_p^N \text{ and } \mathbf{z}_1 = F_0(\rho_0) \cdot F_1(\rho_1) + \mathbf{z}_0\,,\end{gathered} \tag{2}$$

where F_0, F_1 being pseudorandom generators (PRG). Note that while the Setup algorithm for our OLE and VOLE correlation essentially is the identity function, the algorithm might be more complex for other correlations. The reverse-sampling algorithm is defined such that

$$\begin{gathered}(F_1(\rho_1), F_0(\rho_0) \cdot F_1(\rho_1) + \mathbf{z}_0) \leftarrow \mathsf{RSample_{OLE}}(1^\lambda, \mathsf{mk}, 0, (F_0(\rho_0), \mathbf{z}_0)) \text{ and} \\ (F_0(\rho_0), \mathbf{z}_1 - F_0(\rho_0) \cdot F_1(\rho_1)) \leftarrow \mathsf{RSample_{OLE}}(1^\lambda, \mathsf{mk}, 1, (F_1(\rho_1), \mathbf{z}_1)).\end{gathered}$$

Our VOLE correlation is the same as OLE but the value x_1 is a fixed scalar in $\mathbb{F}_p$, i.e., $\mathbf{z}_1 = \mathbf{x}_0 \cdot x_1 + \mathbf{z}_0$. We formally define the reverse-sampleable target correlation with setup $(\mathsf{Setup_{VOLE}}, \mathcal{Y}_{\mathsf{VOLE}})$ of size N over field $\mathbb{F}_p$ as

$$\begin{gathered}\mathsf{mk} = (\rho, x_1) \leftarrow \mathsf{Setup_{VOLE}}(1^\lambda, \rho, x_1)\,, \\ ((F(\rho), \mathbf{z}_0), (x_1, \mathbf{z}_1)) \leftarrow \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, \mathsf{mk}) \quad \text{such that} \\ \mathbf{z}_0 \xleftarrow{\$} \mathbb{F}_p^N \text{ and } \mathbf{z}_1 = F_0(\rho_0) \cdot x_1 + \mathbf{z}_0\,,\end{gathered} \tag{3}$$

```
$\underline{\mathsf{Exp}^{\mathsf{pr\text{-}g}}_{\mathcal{A}}(\lambda):}$
$b \xleftarrow{\$} \{0,1\}, N \leftarrow \eta(\lambda), (\rho_0, \rho_1) \leftarrow \mathcal{A}_0(1^\lambda)$
$\mathsf{mk} \xleftarrow{\$} \mathsf{Setup}(1^\lambda, \rho_0, \rho_1)$
$(\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen_p}(1^\lambda, \rho_0, \rho_1)$
$\textbf{if } b = 0 \textbf{ then } (\mathbf{y}_0, \mathbf{y}_1) \xleftarrow{\$} \mathcal{Y}(1^\lambda, \mathsf{mk})$
$\textbf{else } \mathbf{y}_\sigma \leftarrow \mathsf{PCG.Expand}(\sigma, k_\sigma) \text{ for } \sigma \in \{0,1\}$
$b' \leftarrow \mathcal{A}_1(1^\lambda, \mathbf{y}_0, \mathbf{y}_1), \textbf{ return } b' = b$
```

```
$\underline{\mathsf{Exp}^{\mathsf{key\text{-}ind\text{-}g}}_{\mathcal{A},\sigma}(\lambda):}$
$b \xleftarrow{\$} \{0,1\}, \rho_\sigma \leftarrow \mathcal{A}_0(1^\lambda)$
$\rho^{(0)}_{1-\sigma}, \rho^{(1)}_{1-\sigma} \xleftarrow{\$} \{0,1\}^*, \rho_{1-\sigma} \leftarrow \rho^{(b)}_{1-\sigma}$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCG.Gen_p}(1^\lambda, \rho_0, \rho_1)$
$b' \leftarrow \mathcal{A}_1(1^\lambda, \mathsf{k}_\sigma, \rho^{(0)}_{1-\sigma}) \textbf{ return } b' = b$
```

```
$\underline{\mathsf{Exp}^{\mathsf{sec\text{-}g}}_{\mathcal{A},\sigma}(\lambda):}$
$b \xleftarrow{\$} \{0,1\}, N \leftarrow \eta(\lambda), (\rho_0, \rho_1) \leftarrow \mathcal{A}_0(1^\lambda)$
$\mathsf{mk} \xleftarrow{\$} \mathsf{Setup}(1^\lambda, \rho_0, \rho_1), (\mathsf{k}_0, \mathsf{k}_1) \xleftarrow{\$} \mathsf{PCG.Gen_p}(1^\lambda, \rho_0, \rho_1)$
$\mathbf{y}_\sigma \xleftarrow{\$} \mathsf{PCG.Expand}(\sigma, k_\sigma)$
$\textbf{if } b = 0 \textbf{ then } (\mathbf{y}_{1-\sigma}) \leftarrow \mathsf{PCG.Expand}(1-\sigma, k_{1-\sigma})$
$\textbf{else } \mathbf{y}_{1-\sigma} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{mk}, \sigma, \mathbf{y}_\sigma)$
$b' \leftarrow \mathcal{A}_1(1^\lambda, \mathbf{y}_0, \mathbf{y}_1), \textbf{ return } b' = b$
```

Fig. 1: Security games for reusable PCGs.

where F being a pseudorandom generator (PRG). The reverse-sampling algorithm is defined such that $(x_1, F(\rho) \cdot \mathbf{x}_1 + \mathbf{z}_0) \leftarrow \mathsf{RSample_{VOLE}}(1^\lambda, \mathsf{mk}, 0, (F(\rho), \mathbf{z}_0))$ and $(F(\rho), \mathbf{z}_1 - F(\rho) \cdot x_1) \leftarrow \mathsf{RSample_{VOLE}}(1^\lambda, \mathsf{mk}, 1, (x_1, \mathbf{z}_1))$.

We state PCG constructions realizing these definitions of OLE and VOLE correlations in Appendix D.

4 Threshold Online Protocol

In this section, we present our threshold BBS+ protocol. This protocol yields a signing phase without interaction between the signers and a flexible threshold parameter t.

We show the security of our protocol against a malicious adversary statically corrupting up to $t-1$ parties in the UC framework. We show that our scheme implements a modification of the generic ideal functionality for threshold signature schemes introduced by Canetti et al. [29]. We deliberately chose the generic threshold signature functionality by Canetti et al. [29] over a specific BBS+ functionality such as the one used in [42]. Proving security under a generic threshold functionality enables our threshold BBS+ protocol to be used whenever a threshold signature scheme is required (e.g., for the construction of a more complex protocol such as an anonymous credential system). We present the ideal functionality and discuss the changes with respect to the original version in Appendix H.

Our protocol uses precomputation to accelerate online signing. An intuitive description of the precomputation used is given in Section 1.2. We formally model the precomputation by describing our protocol in a hybrid model where parties can access a hybrid preprocessing functionality $\mathcal{F}_{\mathsf{Prep}}$. Using a hybrid model allows us to abstract from the concrete instantiation of the preprocessing functionality. We present concrete instantiations of $\mathcal{F}_{\mathsf{Prep}}$ in Section 5 and Appendix G.

4.1 Ideal Preprocessing Functionality

The preprocessing functionality consists of two phases. First, the *Initialization* phase samples a private/public key pair. Second, the *Tuple* phase provides correlated tuples upon request. In the second phase, the output values of the honest parties are reverse sampled, given the corrupted parties' outputs. To explicitly model the Tuple phase as non-interactive, we require the simulator to specify a function Tuple during the Initialization. This function defines the corrupted parties' output values in the Tuple phase and is computed first to reverse sample the honest parties' outputs.

Functionality $\mathcal{F}_{\mathsf{Prep}}$

The functionality $\mathcal{F}_{\mathsf{Prep}}$ interacts with parties $P_1, \ldots, P_n$ and ideal-world adversary $\mathcal{S}$. The functionality is parameterized by a threshold parameter t. During the initialization, $\mathcal{S}$ provides a tuple function $\mathsf{Tuple}(\cdot,\cdot,\cdot) \to \mathbb{Z}_p^5$.
Initialization. Upon receiving $(\texttt{init}, \mathsf{sid})$ from all parties,

1. Sample the secret key $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_p$.
2. Send $\mathsf{pk} = (g_2^{\mathsf{sk}})$ to $\mathcal{S}$. Upon receiving $(\texttt{ok}, \mathsf{Tuple}(\cdot,\cdot,\cdot))$ from $\mathcal{S}$, send pk to every honest party.

Tuple. On input $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ from party P_i where $i \in \mathcal{T}$, $\mathcal{T} \subseteq [n]$ of size t do:

– If $(\mathsf{ssid}, \mathcal{T}, \{(a_\ell, e_\ell, s_\ell, \delta_\ell, \alpha_\ell)\}_{\ell\in\mathcal{T}})$ is stored, send $(a_i, e_i, s_i, \delta_i, \alpha_i)$ to P_i.
Else, compute $(a_j, e_j, s_j, \delta_j, \alpha_j) \leftarrow \mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ for every corrupted party P_j where $j \in \mathcal{C} \cap \mathcal{T}$. Next, sample $a, e, s \xleftarrow{\$} \mathbb{Z}_p$ and tuples $(a_j, e_j, s_j, \delta_j, \alpha_j)$ over $\mathbb{Z}_p$ for $j \in \mathcal{H} \cap \mathcal{T}$ such that

$$\begin{gathered}\sum_{\ell\in\mathcal{T}} a_\ell = a \qquad \sum_{\ell\in\mathcal{T}} e_\ell = e \qquad \sum_{\ell\in\mathcal{T}} s_\ell = s \\ \sum_{\ell\in\mathcal{T}} \delta_\ell = a(\mathsf{sk} + e) \qquad \sum_{\ell\in\mathcal{T}} \alpha_\ell = as\end{gathered} \tag{4}$$

Store $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \{(a_\ell, e_\ell, s_\ell, \delta_\ell, \alpha_\ell)\}_{\ell\in\mathcal{T}})$ and send $(\mathsf{sid}, \mathsf{ssid}, a_i, e_i, s_i, \delta_i, \alpha_i)$ to honest party P_i.

Abort. On input $(\texttt{abort}, \mathsf{sid})$ from $\mathcal{S}$, send $\texttt{abort}$ to all honest parties and halt.

4.2 Online Signing Protocol

Next, we formally state our threshold BBS+ protocol. We refer the reader to the technical overview in Section 1.2 for a high-level description of our protocol. Further, we discuss extensions for anonymous credentials systems, blind signing and efficiency improvements in Appendix C.

Construction 1: $\pi_{\mathsf{TBBS+}}$

We describe the protocol from the perspective of an honest party P_i.
Public Parameters. Number of parties n, maximal number of signatures N, size of message arrays k, security threshold t, a bilinear mapping tuple $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, \mathsf{e})$ and randomly sampled $\mathbb{G}_1$ elements $\{h_\ell\}_{\ell\in[0..k]}$. Let $\mathsf{Verify}_{\mathsf{pk}}(\cdot,\cdot)$ be the BBS+ verification algorithm as defined in Appendix A.
KeyGen.

– Upon receiving $(\texttt{keygen}, \mathsf{sid})$ from $\mathcal{Z}$, send $(\texttt{init}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Prep}}$ and receive pk in return.
– Upon receiving $(\texttt{pubkey}, \mathsf{sid})$ from $\mathcal{Z}$ output $(\texttt{pubkey}, \mathsf{sid}, \mathsf{Verify}_{\mathsf{pk}}(\cdot,\cdot))$.

Sign. Upon receiving $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m} = \{m_\ell\}_{\ell\in[k]})$ from $\mathcal{Z}$ with $P_i \in \mathcal{T}$ and no tuple $(\mathsf{sid}, \mathsf{ssid} \mod N, \cdot)$ is stored, perform the following steps:

1. Send $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid} \mod N, \mathcal{T})$ to $\mathcal{F}_{\mathsf{Prep}}$ and receive tuple $(a_i, e_i, s_i, \delta_i, \alpha_i)$.
2. Store $(\mathsf{sid}, \mathsf{ssid}, \mathbf{m})$ and send $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, A_i := (g_1 \cdot \prod_{\ell\in[k]} h_\ell^{m_\ell})^{a_i} \cdot h_0^{\alpha_i}, \delta_i, e_i, s_i)$ to each party $P_j \in \mathcal{T}$.
3. Once $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, A_j, \delta_j, e_j, s_j)$ is received from every party $P_j \in \mathcal{T} \setminus \{P_i\}$,
   (a) compute $e = \sum_{\ell\in\mathcal{T}} e_\ell$, $s = \sum_{\ell\in\mathcal{T}} s_\ell$, $\epsilon = \left(\sum_{\ell\in\mathcal{T}} \delta_\ell\right)^{-1}$, and $A = (\Pi_{\ell\in\mathcal{T}} A_\ell)^\epsilon$.
   (b) If $\mathsf{Verify}_{\mathsf{pk}}(\mathbf{m}, (A, e, s)) = 1$, set $\mathsf{out} = \sigma = (A, e, s)$. Otherwise, set $\mathsf{out} = \texttt{abort}$. Then, output $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \mathsf{out})$.

Verify. Upon receiving $(\texttt{verify}, \mathsf{sid}, \mathbf{m} = \{m_\ell\}_{\ell\in[k]}, \sigma, \mathsf{Verify}_{\mathsf{pk}'}(\cdot,\cdot))$ from $\mathcal{Z}$ output $(\texttt{verified}, \mathsf{sid}, \mathbf{m}, \sigma, \mathsf{Verify}_{\mathsf{pk}'}(\mathbf{m}, \sigma))$.

Remark. While we simplified our UC model to capture the scenario where every signer obtains the final signature, we expect real-world scenarios to have a dedicated client which is the only party to obtain the signature. In the latter case, the signers send the partial signature in Step 2 only to the client and Steps 3a and 3b are performed by the client. We stress that in both cases the communication follows a request-response pattern which is the minimum for MPC protocols. Moreover, note that the $(\texttt{tuple}, \cdot, \cdot, \cdot)$-call to $\mathcal{F}_{\mathsf{Prep}}$ does not involve additional communication when being instantiated based on PCGs or PCFs as done in this work. Using such an instantiation, the $(\texttt{tuple}, \cdot, \cdot, \cdot)$-call is realized by local evaluation of the PCF or local expansion of the PCG so that no interaction between the parties is needed.

Theorem 1. *Assuming the strong unforgeability of BBS+, protocol $\pi_{\mathsf{TBBS+}}$ UC-realizes $\mathcal{F}_{\mathsf{tsig}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model in the presence of malicious adversaries controlling up to $t-1$ parties.*

The proof is given in Appendix I.

5 PCG-based Threshold Preprocessing Protocol

We state our threshold BBS+ signing protocol in Section 4 in a $\mathcal{F}_{\mathsf{Prep}}$-hybrid model. Now, we present an instantiation of the $\mathcal{F}_{\mathsf{Prep}}$ functionality using pseudorandom correlation generators (PCGs). In particular, our $\pi^{\mathsf{PCG}}_{\mathsf{Prep}}$ protocol builds on PCGs for VOLE and OLE correlations. The resulting protocol consists of an interactive *Initialization* and a non-interactive *Tuple* phase, consisting only of the retrieval of stored PCG tuples and additional local computation.

Our preprocessing protocol consists of four steps: the first three are part of the Initialization phase, and the fourth one builds the Tuple phase. First, the parties set up a secret and corresponding public key. For the BBS+ signature scheme, the public key is $\mathsf{pk} = g_2^x$, while the secret key is $\mathsf{sk} = x$, which is secret-shared using Shamir's secret sharing. This procedure constitutes a standard distributed key generation protocol for a DLOG-based cryptosystem. Therefore, we abstract from the concrete instantiation of this protocol and model the key generation as a hybrid functionality $\mathcal{F}_{\mathsf{KG}}$. Second, the parties set up the keys for the PCG instances. The protocol uses two-party PCGs, meaning each pair of parties sets up required instances. We model the PCG key generation as a hybrid functionality $\mathcal{F}^{\mathsf{PCG}}_{\mathsf{Setup}}$. Third, every party expands the local seeds to the required OLE- and VOLE-correlations and store them in their storage. The fourth step constitutes the Tuple phase and is executed by every party in the signer set $\mathcal{T}$ of a signing request.

In this phase party P_i generates $(a_i, e_i, s_i, \delta_i, \alpha_i)$, where the values fulfill correlation (4). For a signing request with ssid, P_i takes the ssid-th component of the previously expanded correlation vectors $\mathbf{a}, \mathbf{e}$ and $\mathbf{s}$ denoted by a_i, e_i, s_i. Note that the a_i values constitute an additive secret sharing of a and the same holds for e and s (cf. (4)). Then, $\sum_{\ell \in \mathcal{T}} \alpha_\ell = as$ can be rewritten as $as = \sum_{\ell \in \mathcal{T}} a_\ell \cdot \sum_{j \in \mathcal{T}} s_j = \sum_{\ell \in \mathcal{T}} \sum_{j \in \mathcal{T}} a_\ell s_j$. Each multiplication $a_\ell s_j$ is equal to the additive shares of an OLE correlation, i.e., $c_1 - c_0 = a_\ell s_j$. The parties use the stored OLE correlations that were expanded in the third step. Note that the parties use again the ssid-th component of the vectors to get consistent values. Finally, party P_i locally adds $a_i s_i$ and the outputs of its PCG expansions to get an additive sharing of as. The same idea works for computing δ_i such that $\sum_{\ell \in \mathcal{T}} \delta_\ell = a(\mathsf{sk} + e) = a\mathsf{sk} + ae$. Note that while the values a, e, s are fresh random values for each signing request, sk is fixed. Therefore, the parties use VOLE correlations to compute $a\mathsf{sk}$ instead of OLE correlations.

Note that party P_i uses PCG instances for computing additive shares of $a_i s_j$ and $a_i s_\ell$ for two different parties P_j and P_ℓ. Since a_i must be the same for both products, we use reusable PCGs so parties can fix a_i over multiple PCG instances. Based on these two requirements, our protocol relies on strong reusable PCGs defined in Section 3.

Key Generation Functionality We abstract from the concrete instantiation of the key generation. Therefore, we state a very simple key generation functionality for discrete logarithm-based cryptosystems similar to the functionality of [71]. The functionality describes a standard distributed key generation for discrete logarithm-based cryptosystems and can be realized by [48, 71] or the key generation phase of [29] or [42].

Functionality $\mathcal{F}_{\mathsf{KG}}$

The functionality is parameterized by the order of the group from which the secret key is sampled p, a generator for the group of the public key g_2, and a threshold parameter t. The key generation functionality interacts with parties $P_1, \ldots, P_n$ and ideal-world adversary $\mathcal{S}$.

Key Generation. Upon receiving $(\texttt{keygen}, \mathsf{sid})$ from every party P_i and $(\texttt{corruptedShares}, \mathsf{sid}, \{\mathsf{sk}_j\}_{j \in \mathcal{C}})$ from $\mathcal{S}$:

1. Sample random polynomial $F \in \mathbb{Z}_p[X]$ of degree $t-1$ such that $F(j) = \mathsf{sk}_j$ for every $j \in \mathcal{C}$.
2. Set $\mathsf{sk} = F(0)$, $\mathsf{pk} = g_2^{\mathsf{sk}}$, $\mathsf{sk}_\ell = F(\ell)$ and $\mathsf{pk}_\ell = g_2^{\mathsf{sk}_\ell}$ for $\ell \in [n]$.
3. Send $(\mathsf{sid}, \mathsf{sk}_i, \mathsf{pk}, \{\mathsf{pk}_\ell\}_{\ell \in [n]})$ to every party P_i.

Setup Functionality The setup functionality gets random values, secret key shares, and partial public keys as input from every party. Then, it first checks if the secret key shares and the partial public keys match and next generates the PCG keys using the random values. Finally, it returns the generated PCG keys to the parties.

In order to provide modularity, we abstract from concrete instantiation by specifying this functionality. Nevertheless, $\mathcal{F}_{\mathsf{Setup}}$ can be instantiated using general-purpose MPC or tailored protocols similar to distributed seed generation protocols from prior work [19, 2]. We leave a formal specification of a tailored protocol as future work.

Functionality $\mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}}$

Let $(\mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Gen}_{\mathsf{p}}, \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand})$ be an rPCG for VOLE correlations and let $(\mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}_{\mathsf{p}}, \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand})$ be an rPCG for OLE correlations. The setup functionality interacts with parties $P_1, \ldots, P_n$.

Setup. Upon receiving $(\texttt{setup}, \mathsf{sid}, \rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}, \mathsf{sk}_i, \{\mathsf{pk}_\ell^{(i)}\}_{\ell\in[n]})$ from every party P_i:

1. Check if $g_2^{\mathsf{sk}_\ell} = \mathsf{pk}_\ell^{(i)}$ for every $\ell, i \in [n]$. If the check fails, send $\texttt{abort}$ to all parties. Else, compute for (P_k, P_j) with $k, j \in [n], k \neq j$:
   (a) $(\mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{k}_{i,j,1}^{\mathsf{VOLE}}) \leftarrow \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Gen}_{\mathsf{p}}(1^\lambda, \rho_a^{(i)}, \mathsf{sk}_j)$,
   (b) $(\mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{i,j,1}^{(\mathsf{OLE},1)}) \leftarrow \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}_{\mathsf{p}}(1^\lambda, \rho_a^{(i)}, \rho_s^{(j)})$, and
   (c) $(\mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{i,j,1}^{(\mathsf{OLE},2)}) \leftarrow \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}_{\mathsf{p}}(1^\lambda, \rho_a^{(i)}, \rho_e^{(j)})$.
2. Send keys $(\mathsf{sid}, \{\mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{k}_{j,i,1}^{\mathsf{VOLE}}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},2)}\}_{j\in[n]\setminus i})$ to party P_i for $i \in [n]$.

PCG-based Preprocessing Protocol In this section, we formally present our PCG-based preprocessing protocol in the $(\mathcal{F}_{\mathsf{KG}}, \mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}})$-hybrid model.

Construction 2: $\pi_{\mathsf{Prep}}^{\mathsf{PCG}}$

Let $(\mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Gen}_{\mathsf{p}}, \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand})$ be an rPCG for VOLE correlations and let $(\mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Gen}_{\mathsf{p}}, \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand})$ be an rPCG for OLE correlations.
We describe the protocol from the perspective of P_i.
Initialization. Upon receiving input $(\texttt{init}, \mathsf{sid})$, do:

1. Send $(\texttt{keygen}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{KG}}$.
2. Upon receiving $(\mathsf{sid}, \mathsf{sk}_i, \mathsf{pk}, \{\mathsf{pk}_\ell^{(i)}\}_{\ell\in[n]})$ from $\mathcal{F}_{\mathsf{KG}}$, sample $\rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)} \in \{0,1\}^\lambda$ and send $(\texttt{setup}, \mathsf{sid}, \rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}, \mathsf{sk}_i, \{\mathsf{pk}_\ell^{(i)}\}_{\ell\in[n]})$ to $\mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}}$.
3. Upon receiving $(\mathsf{sid}, \mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{k}_{j,i,1}^{\mathsf{VOLE}}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},2)})_{j\neq i}$ from $\mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}}$, compute and store for every $j \in [N] \setminus \{i\}$:
   (a) $(\mathbf{a}_i, \mathbf{c}_{i,j,0}^{\mathsf{VOLE}}) = \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(0, \mathsf{k}_{i,j,0}^{\mathsf{VOLE}})$,
   (b) $(\mathsf{sk}_i, \mathbf{c}_{j,i,1}^{\mathsf{VOLE}}) = \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(1, \mathsf{k}_{j,i,0}^{\mathsf{VOLE}})$,
   (c) $(\mathbf{a}_i, \mathbf{c}_{i,j,0}^{(\mathsf{OLE},1)}) = \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)})$,
   (d) $(\mathbf{s}_i, \mathbf{c}_{j,i,1}^{(\mathsf{OLE},1)}) = \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},1)})$,
   (e) $(\mathbf{a}_i, \mathbf{c}_{i,j,0}^{(\mathsf{OLE},2)}) = \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)})$, and
   (f) $(\mathbf{e}_i, \mathbf{c}_{j,i,1}^{(\mathsf{OLE},2)}) = \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},2)})$.
4. Output pk.

Tuple. Upon receiving input $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$, compute:

5. Let $a_i = \mathbf{a}_i[\mathsf{ssid}], e_i = \mathbf{e}_i[\mathsf{ssid}], s_i = \mathbf{s}_i[\mathsf{ssid}], c_{(i,j,0)}^{\mathsf{VOLE}} = \mathbf{c}_{(i,j,0)}^{\mathsf{VOLE}}[\mathsf{ssid}]$, $c_{(j,i,1)}^{\mathsf{VOLE}} = \mathbf{c}_{(j,i,1)}^{\mathsf{VOLE}}[\mathsf{ssid}]$, $c_{(i,j,0)}^{(\mathsf{OLE},d)} = \mathbf{c}_{(i,j,0)}^{(\mathsf{OLE},d)}[\mathsf{ssid}]$ and $c_{(j,i,1)}^{(\mathsf{OLE},d)} = \mathbf{c}_{(j,i,1)}^{(\mathsf{OLE},d)}[\mathsf{ssid}]$ for $j \in \mathcal{T} \setminus \{i\}$ and $d \in \{1,2\}$.
6. Compute
   $\delta_i = a_i(e_i + L_{i,\mathcal{T}}\mathsf{sk}_i) + \sum_{j\in\mathcal{T}\setminus\{i\}} \left(L_{i,\mathcal{T}} c_{j,i,1}^{\mathsf{VOLE}} - L_{j,\mathcal{T}} c_{i,j,0}^{\mathsf{VOLE}} + c_{j,i,1}^{(\mathsf{OLE},2)} - c_{i,j,0}^{(\mathsf{OLE},2)}\right)$
7. Compute $\alpha_i = a_i s_i + \sum_{j\in\mathcal{T}\setminus\{i\}} \left(c_{j,i,1}^{(\mathsf{OLE},1)} - c_{i,j,0}^{(\mathsf{OLE},1)}\right)$
8. Output $(\mathsf{sid}, \mathsf{ssid}, a_i, e_i, s_i, \delta_i, \alpha_i)$.

Theorem 2. *Let $\mathsf{PCG}_{\mathsf{VOLE}}$ be an rPCG for VOLE correlations and let $\mathsf{PCG}_{\mathsf{OLE}}$ be an rPCG for OLE correlations. Then, protocol $\pi_{\mathsf{Prep}}^{\mathsf{PCG}}$ UC-realizes $\mathcal{F}_{\mathsf{Prep}}$ in the $(\mathcal{F}_{\mathsf{KG}}, \mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}})$-hybrid model in the presence of malicious adversaries controlling up to $t-1$ parties.*

We state our simulator, a proof sketch and the full indistinguishability proof in Appendix J, K, and L.

6 Evaluation

In the following, we present the evaluation of the online and the offline phase of our protocol. As [67] published an optimization of the BBS+ signature scheme concurrent to our work, we repeat our evaluation for an optimized version of our protocol and present the results in Appendix N.

Parameters. In the following, we denote the security parameter by λ, the number of servers by n, the security threshold by t, the size of the signed message arrays by k, the number of generated precomputation tuples by N, the order of the elliptic curve's groups $\mathbb{G}_1$ and $\mathbb{G}_2$ by p and assume PCGs based on the Ring LPN problem with static leakage and security parameters c and τ, i.e., the $R^c\text{-}\mathsf{LPN}_{p,\tau}$ assumption.[3] This assumption is common to state-of-the-art PCG instantiations for OLE correlations [19].

6.1 Online, Signing Request-Dependent Phase

We evaluate the online, signing request-dependent phase by implementing the protocol, running benchmarks, and reporting the runtime and the communication complexity. For comparison, we also implement and benchmark the non-threshold BBS+ signing algorithm. We open-source our prototype implementation to foster future research in this area.[4]

Implementation and experimental setup. Our implementation and benchmarks of the online phase are written in Rust and based on the BLS12_381 curve.[5] Note, since the BLS12_381 curve defines an elliptic curve, we use the additive group notation in the following. This is in contrast to the multiplicative group notation used in the protocol description. Our code, including the benchmarks and rudimentary tests, comprises 1,400 lines. We compiled our code using rustc 1.68.2 (9eb3afe9e).

For our benchmarks, we split the protocol into four phases: *Adapt* (Steps 6 and 7 of protocol $\pi^{\mathsf{PCG}}_{\mathsf{Prep}}$), *Sign* (Step 2 of $\pi_{\mathsf{TBBS+}}$), *Reconstruct* (Step 3a of $\pi_{\mathsf{TBBS+}}$) and *Verify* (Step 3b of $\pi_{\mathsf{TBBS+}}$). Adapt and Sign are executed by the servers. Reconstruct and Verify are executed by the client. Together, these phases cover the whole online signing protocol. The runtime of our protocol is influenced by the security threshold t and the message array size k. We perform benchmarks for $2 \leq t \leq 30$ and $1 \leq k \leq 50$. The range for parameter t is chosen to provide comparability with [42] and we deem $k \leq 50$ a realistic setting for the use-cases of credential certificates. Moreover, both ranges illustrate the trend for increasing parameters. The influence of the total number of servers n is insignificant to non-existent. Our benchmarks do not account for network latency, which heavily depends on the location of clients and servers. Network latency, in our protocol, incurs the same overhead as in the non-threshold setting. It can be incorporated by adding the round-trip time of messages up to 2KB over the client's (slowest) server connection to the total runtime. As the online phase of our protocol is non-interactive, we benchmark servers and clients individually. We execute all benchmarks on a single machine with a 14-core Intel Xeon Gold 5120 CPU @ 2.20GHz processor and 64GB of RAM. We repeat each benchmark 100 times to account for statistical deviations and report the average. For comparability, we report the runtime of basic arithmetic operations in Table 1 in Appendix M.

[3] For 128-bit security and $N = 2^{20}$, [19] reports $(c, \tau) \in \{(2, 76), (4, 16), (8, 5)\}$.

[4] `https://github.com/AppliedCryptoGroup/NI-Threshold-BBS-Plus-Code`

[5] We have used [4] for all curve operations.

Experimental Results. We report the results of our benchmarks in Figure 2. These results reflect our expectations as outlined in the following. The *Adapt* phase transforming PCF/PCG outputs to signing request-dependent presignatures involves only field operations and is much faster than the other phases for small t. The runtime increase for larger t stems from the number of field operations scaling quadratically with the number of signers. Signers have to compute a LaGrange coefficient for each other signer. The computation of the LaGrange coefficient scales with t as well. The *Sign* phase requires the servers to compute $k+2$ scalar multiplications in $\mathbb{G}_1$, each taking 100 times more time than the slowest field operation (cf. Appendix M). The *Reconstruct* phase involves a single $\mathbb{G}_1$ scalar multiplication, field operations, and $\mathbb{G}_1$ additions, depending on the threshold t. The scalar multiplication, being responsible for more than 90% of the phase's runtime for $t \leq 30$, dominates the cost of this phase. The *Verify* phase requires the client to compute two pairing operations, a single scalar multiplication in $\mathbb{G}_2$, $k+1$ scalar multiplications $\mathbb{G}_1$, and multiple additions in $\mathbb{G}_1$ and $\mathbb{G}_2$. The pairing operations and the scalar multiplication in $\mathbb{G}_2$ are responsible for the constant costs visible in the graph. The scalar multiplications in $\mathbb{G}_1$ cause the linear increase. The influence of $\mathbb{G}_1$ and $\mathbb{G}_2$ additions is insignificant because they take at most 1.4% of scalar multiplication in $\mathbb{G}_1$. The *Total* runtime mainly depends on the size of the signed message array due to the scalar multiplications in the signing and verification step. The number of signers, t, has only a minor influence on the online runtime; increasing the number of signers from 2 to 30 increases the runtime by $1.14\% - 5.52\%$. Following, the online protocol can essentially tolerate any number of servers as long as the preprocessing, which is expected to scale worse, can be instantiated efficiently for the number of servers and the storage complexity of the generated preprocessing material does not exceed the servers' capacities (cf. Section 6.2).

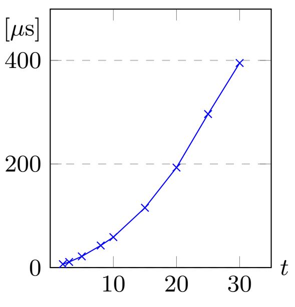
To measure the *overhead of thresholdization*, we compare the runtime of our online protocol to the runtime of signature creation in the non-threshold setting in Figure 3. The overhead of our online protocol consists only of a single scalar multiplication in $\mathbb{G}_1$, assuming that clients also verify received signatures in the non-threshold setting. This observation reflects our protocol pushing all the overhead of the distributed signing to the offline phase.

Communication complexity. The client has to send one signing request of size $(k \cdot \lceil \log p \rceil) + (t \cdot \lceil \log n \rceil)$ bits to each of the t selected servers. By deriving the signer set via a random oracle, we can reduce the size of the request to $(k \cdot \lceil \log p \rceil)$. Each selected server has to send a partial signature of size $(3\lceil \log p \rceil + |\mathbb{G}_1|)$. In case of the BLS12_381 curve, $\lceil \log p \rceil$ equals 381 bits whereas $|\mathbb{G}_1|$ equals 762 bits. Parties can also encode $\mathbb{G}_1$ elements with 381 bits by only sending the x-coordinate of the curve point and requiring the sender to compute the y-coordinate itself.

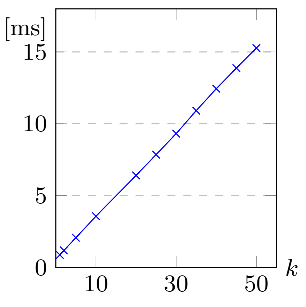
Note that our UC functionality models a scenario where every signer obtains the final signature. Therefore, the partial signatures are sent to all other signers. However, by incorporating a dedicated client into the model, the signers can send the partial signatures only to the client. While we expect this to be sufficient for real-life settings, it makes the model messier. We emphasize that this request-response behavior is the minimum interaction for MPC protocols. As there is no interaction between the servers, this setting is referred to as non-interactive in the literature [29, 2].

6.2 Offline, Signing Request-Independent Phase

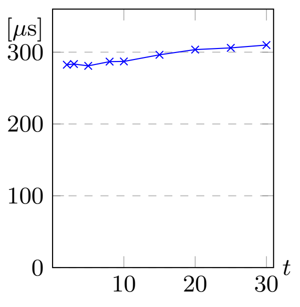
For the offline, signing request-independent phase, we focus on the PCG-based precomputation as PCFs lack efficient instantiations. We compute the communication complexity of the distributed seed generation, the storage complexity of the generated seeds and expanded tuples, and computation complexity of the seed expansion phase. We further implement the seed expansion of the PCGs (Step 3 of protocol $\pi^{\mathsf{PCG}}_{\mathsf{Prep}}$), run benchmarks and report the runtime.



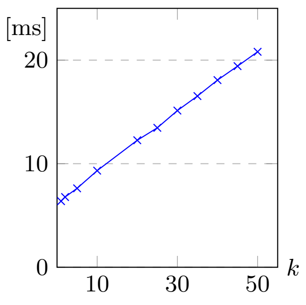
(a) Adapt (Server).



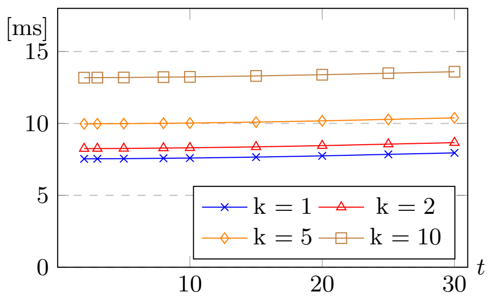
(b) Sign (Server).



(c) Reconstruct (Client).



(d) Verify (Client).



(e) Total.

Fig. 2: The runtime of individual protocol phases (a)-(d) and the total online protocol (e). The *Adapt* phase, describing Steps 5 and 6 of protocol π_{Prep}, and the *Reconstruct* phase, describing Step 3a of $\pi_{\mathsf{TBBS+}}$, depend on security threshold t. The *Sign* phase, describing Step 2 of $\pi_{\mathsf{TBBS+}}$, and the signature verification, describing Step 3b of $\pi_{\mathsf{TBBS+}}$, depend on the message array size k.

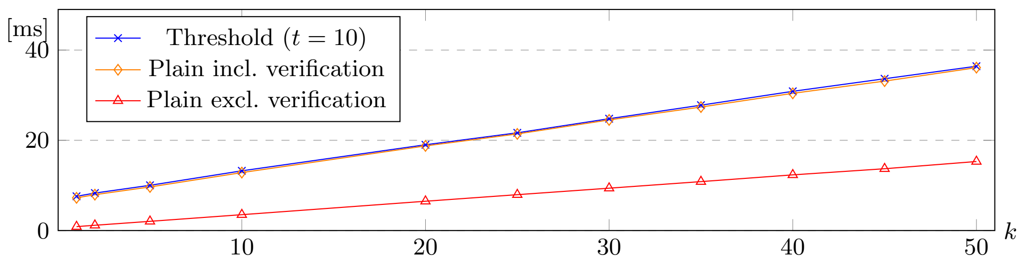


Fig. 3: The total runtime of our online protocol compared to plain, non-threshold signing with and without signature verification in dependence of k. The number of signers t is insignificant (cf. Figure 2e).

Experimental setup. Our implementation[6] and benchmarks are written in Go. Our code, including the benchmarks and rudimentary tests, comprises 5 467 lines. We compiled our code using go 1.21.3. Again, we execute all benchmarks on machines with a 14-core Intel Xeon Gold 5120 CPU @ 2.20GHz processor and 64GB of RAM. Due to the complexity of the benchmarks and the high amount of repetition within a single protocol run, we execute the benchmarks for each choice of parameters just once.

The runtime of the seed expansion is influenced by the number of parties n, the number of generated precomputation tuples N and the Module Ring LPN security parameters (c, τ). For security parameters we fix $c = 4$ and $\tau = 16$ which corresponds to 128-bit security [19]. We compute over a cyclotomic ring as proposed by [19] and fix the prime p to be the order of the BLS12_381 curve. Our tests have shown that this choice of parameters yields the best performance of the possible choices for the same security level. For the number of parties, we consider $2 \leq n \leq 10$.[7] Further, we consider both the t-out-of-n setting and the n-out-of-n setting as the latter has tremendous potential for optimization as discussed below. For the number of generated triples, we consider $N \in [2^{11}, \ldots, 2^{16}]$. For scenarios with less parties, we also consider $N \in [2^{17}, 2^{18}, 2^{19}]$.

Our benchmarks cover the seed expansion phase of all required PCGs (Step 3 of $\pi^{\mathsf{PCG}}_{\mathsf{Prep}}$). As our PCG instantiations compute over a ring, they also return ring elements each representing an array of N field elements. For example, for a batch of N OLE correlations $\mathbf{a} \cdot \mathbf{b} = \mathbf{c} + \mathbf{d}$ ($\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{Z}_p^N$), the PCG actually returns four degree-N polynomials $A \cdot B = C + D$. By choosing the ring appropriately (cf. [19]), each polynomial can be split into N independent OLE correlations over $\mathbb{Z}_p$. This step does not need to happen in a batch but can be done individually. We report the computation time of the PCG seed expansion, yielding the ring elements, and the time to split a single OLE correlation over $\mathbb{Z}_p$ from a ring element, separately.

n-out-of-n vs. t-out-of-n. The runtime of the seed expansion strongly depends on whether we consider a t-out-of-n or a n-out-of-n setting. To understand this dependency, recall the basic concepts of PCGs (cf. PCG constructions in Appendix D). Parties first compute the desired correlation with sparse polynomials as input values. Then, they expand these preliminary sparse correlations to real random correlations by applying an LPN-based randomization. In our protocol, parties do this for each individual OLE- or VOLE-correlation and then combine the real correlations to get the final precomputation tuples. However, in a real implementation parties can first combine the sparse correlations and then apply the LPN-based randomization, effectively reducing the amount of randomization operations. In the t-out-of-n setting, the signer set is only known during the online phase, i.e., after the randomization step. As the combination itself largely depends on the signer set, parties can only perform the combination steps that are independent of the set. In the n-out-of-n setting, the signer set is already known during the offline phase, i.e., every party has to sign. Parties can therefore perform most of the combinations before randomization. More precisely, in the n-out-of-n setting, each party has to perform six randomizations and split five polynomials, while in the t-out-of-n setting each party performs $3 + 4 \cdot (n - 1)$ randomizations and splits just as many polynomials.

Experimental results. In Figure 4 and Figure 5, we display the computation time per signature of the PCG expansion in the n-out-of-n setting and the t-out-of-n setting. The computation time per signature increases superlinear with the number of signatures (note that the x-axis has a logarithmic scale) and linear with the number of parties n. This is due to the fact that the seed expansion requires multiplication of degree-N polynomials. We perform the multiplication via

[6] `https://github.com/leandro-ro/Threshold-BBS-Plus-PCG`

[7] The only prior work implementing the seed expansion [2] is restricted to $n \in \{2, 3\}$.

the Fast Fourier Transformation which scales superlinear with the degree of the polynomial. Both graphs show that the runtime increases with the number N. Nevertheless, as the correlations are expanded from small keys, a large batch size N benefit from the sublinear communication complexity in N of a distributed seed generation.

We further benchmarked the computation time to extract one of N field elements from a degree-N polynomial. The results range from 0.1ms for $N = 2^{11}$ to 8.6ms for $N = 2^{19}$. This step essentially represents a polynomial evaluation executed via the Horner's method which explains the linear increase in the computation time.

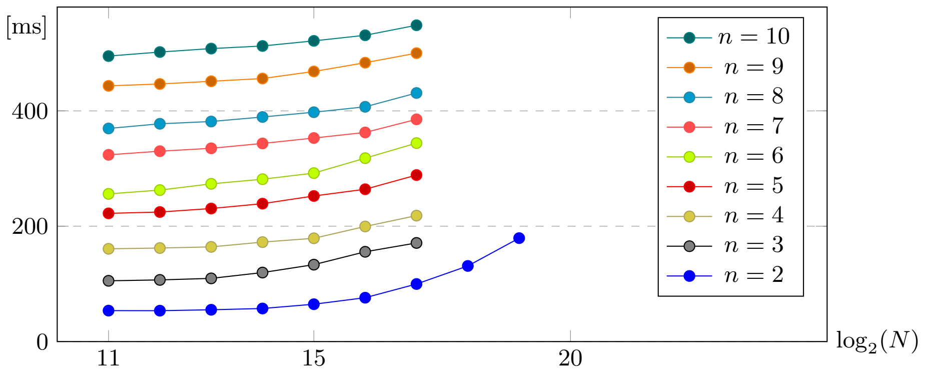


Fig. 4: Computation time of the seed expansion of all required PCGs in the n-out-of-n setting for different committee sizes ($n \in \{2, \dots, 10\}$) dependent on the number of generated precomputation tuples N.

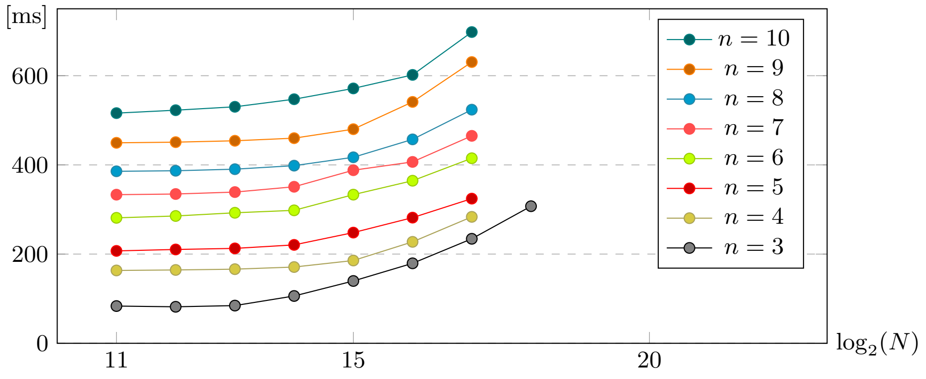


Fig. 5: Computation time of the seed expansion of all required PCGs in the t-out-of-n setting for different committee sizes ($n \in \{2, \dots, 10\}$) dependent on the number of generated precomputation tuples N.

Complexity analysis. Existing fully distributed PCG constructions for OLE-correlations [19, 2] do not separate between the PCG seed generation and the PCG evaluation phase. Instead, they merge both phases into one distributed protocol. These distributed protocols make use of secret sharing-based general-purpose MPC protocols optimized for different kinds of operations (binary [59], field [39, 38], or elliptic curve [37]) as well as a special-purpose protocol for the computation of a two-party distributed point function (DPF) presented in [19]. As the PCG-generated preprocessing material utilized in [2] shows similarities to the material required by our online signing protocol, we derive a distributed PCG protocol for our setting from theirs and analyze the communication complexity accordingly. The analysis yields that the communication complexity of a PCG-based preprocessing instantiating our offline protocol is dominated by

$$26(nc\tau)^2 \cdot (\log N + \log p) + 8n(c\tau)^2 \cdot \lambda \cdot \log N$$

bits of communication per party.

Instead of merging the PCG setup with the PCG evaluation in one setup protocol, it is also possible to generate the PCG seeds first, either via a trusted party or another dedicated protocol, and execute the expansion at a later point in time, e.g., when the next batch of presignatures is required. In this scenario, each server stores seeds with a size of at most

$$\begin{aligned}&\log p + 3c\tau \cdot (\lceil \log p \rceil + \lceil \log N \rceil)\\ &+2(n-1)\cdot c\tau \cdot (\lceil \log N \rceil \cdot (\lambda+2) + \lambda + \lceil \log p \rceil)\\ &+4(n-1)\cdot (c\tau)^2 \cdot (\lceil \log 2N \rceil \cdot (\lambda+2) + \lambda + \lceil \log p \rceil)\end{aligned}$$

bits if the PCGs are instantiated according to [19].

When instantiating the precomputation with PCGs, servers must evaluate all of the PCGs' outputs at once. The resulting precomputation material occupies

$$\log p \cdot N \cdot (3 + 6 \cdot (n-1))$$

bits of storage. In [2], the authors report $N = 94\,019$ as a reasonable parameter for a PCG-based setup protocol. In [19], the authors base their analysis on $N = 2^{20} = 1\,048\,576$. To efficiently apply Fast Fourier Transformation algorithms during the seed expansion, it is necessary to choose N such that it divides $p-1$. Figure 6 reports the storage complexity depending on the number of servers n for different N. Note that the dependency on the number of servers n stems from the fact that we support any threshold $t \leq n$. In a n-out-of-n settings, servers execute Steps 6 and 7 of $\pi^{\mathsf{PCG}}_{\mathsf{Prep}}$ during the preprocessing, and hence, only store $\log p \cdot 5N$ bits of preprocessing material.

The computation cost of the seed expansion is dominated by the ones of the PCGs for OLE correlations. In [19], the authors report the computation complexity of expanding a seed of an OLE PCG to involve at most $N(ct)^2(4 + 2\lfloor \log(p/\lambda) \rfloor)$ PRG operations and $O(c^2 N \log N)$ operations in $\mathbb{Z}_p$. In our protocol, each server P_i has to evaluate 4 OLE-generating PCGs for each other server P_j; one for each cross term $(a_i \cdot e_j)$, $(a_j \cdot e_i)$, $(a_i \cdot s_j)$, and $(a_j \cdot s_i)$. It follows that the seed expansion in our protocol is dominated by

$$4 \cdot (n-1) \cdot (4 + 2\lfloor \log(p/\lambda) \rfloor) \cdot N \cdot (c\tau)^2$$

PRG evaluations and $O(nc^2 N \log N)$ operations in $\mathbb{Z}_p$.

6.3 Comparison to [42]

Independently of our work, [42] presented the first t-out-of-n threshold BBS+ protocol. This protocol incorporates a lightweight setup independent of the number of generated signatures but

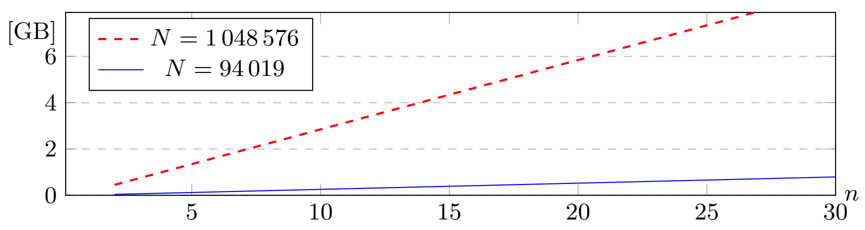
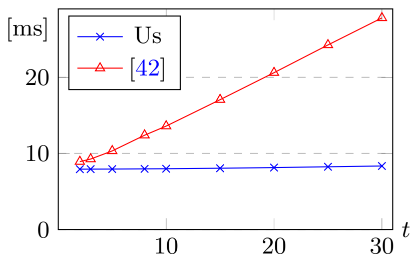
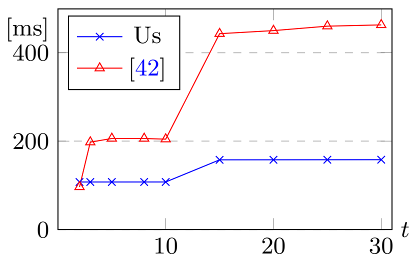


Fig. 6: Storage complexity of the precomputation material required for $N \in \{94\,019, 1\,048\,576\}$ signatures depending on the number of servers n.



(a) LAN.



(b) WAN.

Fig. 7: Runtime of the signing protocol of [42] compared to the network adjusted runtime of our signing protocol in the LAN and WAN setting.

requires an interactive signing protocol. In contrast, our scheme offers a trade-off that provides an efficient, non-interactive online phase at the cost of a more complex offline phase. This trade-off aims to minimize the time it takes to answer a signing request. To show that our online phase's efficiency indeed benefits from the costly preprocessing, we compare our online signing phase to their interactive signing protocol.[8] We stress that the advantage of our online phase comes at the cost of a significantly more complex offline phase. However, our online phase is independent of the concrete instantiation of the offline phase. In particular, less memory-consuming but more communication-intensive instantiations, e.g., based on oblivious transfer or somewhat homomorphic encryption, are also possible.

As our implementation, their implementation is in Rust and based on the BLS12_381 curve. When comparing the benchmarking machines, $\mathbb{G}_1$ and $\mathbb{G}_2$ scalar multiplications are $20 - 30\%$ faster on our machine, while signature verifications are 20% faster on their machine. Although not explicitly stated, the numbers strongly indicate the choice $k = 1$ in [42]; the reported runtime of non-threshold BBS+ signing is slightly larger than three $\mathbb{G}_1$ scalar multiplications. Due to the interactivity of their protocol, their benchmarks incorporate network delays for different settings (LAN, WAN). We add network delays to our results to compare our benchmarks to theirs. All machines used in their evaluation are *Google Cloud c2d-standard-4* instances. In the LAN setting, all instances are located at the us-east1-c zone. [40] reports a LAN latency of 0.146 ms for this zone. We add a delay of 0.3 ms to our results. In the WAN setting, the first 12 instances in their benchmarks are located in the US, while other machines are in Europe or the US. According to [52], we add 100 ms to our results for $t < 13$ and 150 ms for $t \geq 13$.

In Figure 7, we compare the runtime, including latency, of our online signing protocol to the runtimes reported in [42] for the LAN and the WAN setting. The graphs show that our protocol

[8] We thank the authors of [42] for sharing concrete numbers of their evaluation.

outperforms the one of [42] in both settings for every number of servers. The only exception is the runtime for $t = 2$ in the WAN setting. This exception seems caused by an unusually low connection latency between the first two servers and the client in [42]. The overhead of [42] is mainly caused by the two additional rounds of cross-server interaction. This overhead rises with the number of servers as each server has to communicate with each other servers and is especially severe in the WAN setting.

Due to the high efficiency and non-interactivity of our online phase, our protocol is more suited for settings where servers have a sufficiently long setup interval and storage capacities to deal with the complexity of the preprocessing phase. On the other hand, the protocol of [42] is more suited for use cases with more lightweight servers, especially in a LAN environment where the network delay of the additional communication is less significant.

Acknowledgments

The first, third, and fifth authors were supported by the German Research Foundation (DFG) CRC 1119 CROSSING (project S7), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, and by the European Research Council (ERC) under the European Union's Horizon 2020 and Horizon Europe research and innovation programs (grant CRYPTOLAYER-101044770). The second author was partially supported 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 the Algorand Foundation and the United States-Israel Binational Science Foundation (BSF) through Grant No. 2020277.

References

1. Masayuki Abe and Tatsuaki Okamoto. Provably secure partially blind signatures. In *CRYPTO*, 2000.
2. Damiano Abram, Ariel Nof, Claudio Orlandi, Peter Scholl, and Omer Shlomovits. Low-bandwidth threshold ECDSA via pseudorandom correlation generators. In *IEEE SP*, 2022.
3. Damiano Abram and Peter Scholl. Low-communication multiparty triple generation for spdz from ring-lpn. In *PKC*, 2022.
4. Algorand. BLS12-381 Rust crate. `https://github.com/algorand/pairing-plus`. (Accessed on 04/17/2024).
5. Man Ho Au, Willy Susilo, and Yi Mu. Constant-size dynamic k-TAA. In *SCN*, 2006.
6. Jean-Philippe Aumasson, Adrian Hamelink, and Omer Shlomovits. A survey of ECDSA threshold signing. *IACR Cryptol. ePrint Arch.*, 2020.
7. Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *PODC*, 1989.
8. Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.
9. Greg Bernstein and Manu Sporny. Data integrity bbs cryptosuites v1.0. `https://w3c.github.io/vc-di-bbs/`, April 2024. (Accessed on 04/17/2024).
10. Johannes Blömer, Jan Bobolz, Denis Diemert, and Fabian Eidens. Updatable anonymous credentials and applications to incentive systems. In *CCS*, 2019.
11. Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In *CRYPTO*, 2004.
12. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In *CRYPTO*, 2001.
13. Alexandre Bouez and Kalpana Singh. One round threshold ECDSA without roll call. In *CT-RSA*, 2023.

14. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In *CCS*, 2018.
15. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Nicolas Resch, and Peter Scholl. Correlated pseudorandomness from expand-accumulate codes. In *CRYPTO*, 2022.
16. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *CCS*, 2019.
17. 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.
18. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Correlated pseudorandom functions from variable-density LPN. In *FOCS*, 2020.
19. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In *CRYPTO*, 2020.
20. Ernie Brickell and Jiangtao Li. A pairing-based DAA scheme further reducing TPM resources. In *TRUST*, 2010.
21. Ernie Brickell and Jiangtao Li. Enhanced privacy ID from bilinear pairing for hardware authentication and attestation. *Int. J. Inf. Priv. Secur. Integr.*, 2011.
22. Jan Camenisch. Anonymous credentials: Opportunities and challenges. In *SEC*, 2006.
23. Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation using the strong diffie hellman assumption revisited. In *TRUST*, 2016.
24. Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: Definitions and practical constructions. In *ASIACRYPT*, 2015.
25. Jan Camenisch, Stephan Krenn, Anja Lehmann, Gert Læssøe Mikkelsen, Gregory Neven, and Michael Østergaard Pedersen. Formal treatment of privacy-enhancing credential systems. In *SAC*, 2015.
26. Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *EUROCRYPT*, 2001.
27. Jan Camenisch and Anna Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *CRYPTO*, 2004.
28. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
29. Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *CCS*, 2020.
30. Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA. In *PKC*, 2020.
31. Guilhem Castagnos, Fabien Laguillaumie, and Ida Tucker. Threshold linearly homomorphic encryption on $\mathbf{Z}/2^k\mathbf{Z}$. In *ASIACRYPT*, 2022.
32. David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Commun. ACM*, 1985.
33. Lidong Chen. Access with pseudonyms. In *Cryptography: Policy and Algorithms*, 1995.
34. Liqun Chen. A DAA scheme requiring less TPM resources. In *Information Security and Cryptology*, 2009.
35. Hien Chu, Paul Gerhart, Tim Ruffing, and Dominique Schröder. Practical schnorr threshold signatures without the algebraic group model. In *CRYPTO*, 2023.
36. Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. Silver: Silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *CRYPTO*, 2021.
37. Anders Dalskov, Marcel Keller, Claudio Orlandi, Kris Shrishak, and Haya Shulman. Securing dnssec keys via threshold ecdsa from generic mpc, 2020.
38. 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.
39. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.
40. Rick Jones Derek Phanekham. How much is google cloud latency (gcp) between regions? `https://cloud.google.com/blog/products/networking/using-netperf-and-ping-to-measure-network-latency`, June 2020. (Accessed on 04/17/2024).

41. Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Authenticated garbling from simple correlations. In *CRYPTO*, 2022.
42. Jack Doerner, Yash Kondi, Eysa Lee, abhi shelat, and LakYah Tyner. Threshold bbs+ signatures for distributed anonymous credential issuance. In *IEEE SP*, 2023.
43. Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ECDSA from ECDSA assumptions: The multiparty case. In *SP*, 2019.
44. Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. *J. Cryptol.*, 1996.
45. Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *CRYPTO*, 2018.
46. Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ECDSA with fast trustless setup. In *CCS*, 2018.
47. Rosario Gennaro, Steven Goldfeder, and Bertrand Ithurburn. Fully distributed group signatures, 2019.
48. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *EUROCRYPT*, 1999.
49. Chelsea Komlo and Ian Goldberg. FROST: flexible round-optimized schnorr threshold signatures. In *SAC*, 2020.
50. Yashvanth Kondi, Bernardo Magri, Claudio Orlandi, and Omer Shlomovits. Refresh when you wake up: Proactive threshold wallets with offline devices. In *SP*, 2021.
51. Yashvanth Kondi, Claudio Orlandi, and Lawrence Roy. Two-round stateless deterministic two-party schnorr signatures from pseudorandom correlation functions. *IACR Cryptol. ePrint Arch.*, 2023.
52. Chandan Kumar. How much is google cloud latency (gcp) between regions? https://geekflare.com/google-cloud-latency/, March 2022. (Accessed on 04/17/2024).
53. Yehuda Lindell. Fast secure two-party ECDSA signing. In *CRYPTO*, 2017.
54. Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *CCS*, 2018.
55. Tobias Looker, Vasilis Kalos, Andrew Whitehead, and Mike Lodder. The BBS Signature Scheme. Internet-Draft draft-irtf-cfrg-bbs-signatures-02, Internet Engineering Task Force, March 2023. (Work in Progress).
56. Anna Lysyanskaya, Ronald L. Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In *SAC*, 1999.
57. MATTR. mattrglobal/bbs-signatures: An implementation of bbs+ signatures for node and browser environments. https://github.com/mattrglobal/bbs-signatures. (Accessed on 04/17/2024).
58. Microsoft. microsoft/bbs-node-reference: Typescript/node reference implementation of bbs signature. https://github.com/microsoft/bbs-node-reference. (Accessed on 04/17/2024).
59. 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.
60. Claudio Orlandi, Peter Scholl, and Sophia Yakoubov. The rise of paillier: Homomorphic secret sharing and public-key silent OT. In *EUROCRYPT*, 2021.
61. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, 1991.
62. David Pointcheval and Olivier Sanders. Short randomizable signatures. In *CT-RSA*, 2016.
63. Alfredo Rial and Ania M. Piotrowska. Security analysis of coconut, an attribute-based credential scheme with threshold issuance. *IACR Cryptol. ePrint Arch.*, 2022.
64. Adi Shamir. How to share a secret. *Commun. ACM*, 1979.
65. Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *IMA*, 2019.
66. Alberto Sonnino, Mustafa Al-Bassam, Shehar Bano, Sarah Meiklejohn, and George Danezis. Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In *NDSS*, 2019.
67. Stefano Tessaro and Chenzhi Zhu. Revisiting BBS signatures. In *EUROCRYPT*, 2023.
68. Trinsic. Credential api - documentation. https://docs.trinsic.id/reference/services/credential-service/. (Accessed on 04/17/2024).

69. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*, 2017.
70. Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.
71. Douglas Wikström. Universally composable DKG with linear number of exponentiations. In *SCN*, 2004.
72. Harry W. H. Wong, Jack P. K. Ma, Hoover H. F. Yin, and Sherman S. M. Chow. Real threshold ECDSA. In *NDSS*, 2023.
73. Zuoxia Yu, Man Ho Au, and Rupeng Yang. Accountable anonymous credentials. In *Advances in Cyber Security: Principles, Techniques, and Applications*. 2019.

A The BBS+ Signature Scheme

Let k be the size of the message arrays, $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, g_1, g_2, \mathsf{e})$ be a bilinear mapping tuple and $\{h_\ell\}_{\ell \in [0..k]}$ be random elements of $\mathbb{G}_1$. The BBS+ signature scheme is defined as follows:

- $\mathsf{KeyGen}(\lambda)$: Sample $x \xleftarrow{\$} \mathbb{Z}_p^*$, compute $y = g_2^x$, and output $(\mathsf{pk}, \mathsf{sk}) = (y, x)$.
- $\mathsf{Sign}_{\mathsf{sk}}(\{m_\ell\}_{\ell \in [k]} \in \mathbb{Z}_p^k)$: Sample $e, s \xleftarrow{\$} \mathbb{Z}_p$, compute $A := (g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$ and output $\sigma = (A, e, s)$.
- $\mathsf{Verify}_{\mathsf{pk}}(\{m_\ell\}_{\ell \in [k]} \in \mathbb{Z}_p^k, \sigma)$: Output 1 iff $\mathsf{e}(A, y \cdot g_2^e) = \mathsf{e}(g_1 \cdot h_0^s \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell}, g_2)$

The BBS+ signature scheme is proven strong unforgeable under the q-strong Diffie Hellman (SDH) assumption for pairings of type 1, 2, and 3 [5, 23, 67]. Intuitively, strong unforgeability states that the attacker is not possible to come up with a forgery even for messages that have been signed before. We refer to [67] for further details.

Optimized scheme of Tessaro and Zhu [67] Concurrently to our work, Tessaro and Zhu showed an optimized version of the BBS+ signatures, reducing the signature size. In their scheme, the signer samples only one random value, $e \xleftarrow{\$} \mathbb{Z}_p$, computes $A := (g_1 \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{\frac{1}{x+e}}$, and outputs $\sigma = (A, e)$. The verification works as before, with the only difference that the term h_0^s is removed. Note that if the first message m_1 is sampled randomly, then the short version is equal to the original version. While we describe our protocol in the original BBS+ scheme by Au et al. [5], we elaborate on the influence of [67] on our evaluation in Appendix N.

B Universal Composability Framework ([28])

We formally model and prove the security of our protocols in the Universal Composability framework (UC). The framework was introduced by Canetti in 2001 [28] to analyze the security of protocols formally. The universal composability property guarantees the security of a protocol holds even under concurrent composition. We give a brief intuition and defer the reader to [28] for all details.

Like simulation-based proofs, the framework differentiates between real-world and ideal-world execution. The real-world execution consists of n parties $P_1, \ldots, P_n$ executing protocol π, an adversary $\mathcal{A}$, and an environment $\mathcal{Z}$. All parties are initialized with security parameter λ and a random tape, and $\mathcal{Z}$ runs on some advice string z. In this work, we consider only static corruption, where the adversary corrupts parties at the onset of the execution. After corruption, the adversary may instruct the corrupted parties to deviate arbitrarily from the protocol specification. The environment provides inputs to the parties, instructs them to continue the execution of π, and receives outputs from the parties. Additionally, $\mathcal{Z}$ can interact with the adversary.

The real-world execution finishes when $\mathcal{Z}$ stops activating parties and outputs a decision bit. We denote the output of the real-world execution by $\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(\lambda, z)$.

The ideal-world execution consists of n dummy parties, an ideal functionality $\mathcal{F}$, an ideal adversary $\mathcal{S}$, and an environment $\mathcal{Z}$. The dummy parties forward messages between $\mathcal{Z}$ and $\mathcal{F}$, and $\mathcal{S}$ may corrupt dummy parties and act on their behalf in the following execution. $\mathcal{S}$ can also interact with $\mathcal{F}$ directly according to the specification of $\mathcal{F}$. Additionally, $\mathcal{Z}$ and $\mathcal{S}$ may interact. The goal of $\mathcal{S}$ is to simulate a real-world execution such that the environment cannot tell apart if it is running in the real or ideal world. Therefore, $\mathcal{S}$ is also called the simulator.

Again, the ideal-world execution ends when $\mathcal{Z}$ outputs a decision bit. We denote the output of the ideal-world execution by $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\lambda, z)$.

Intuitively, a protocol is secure in the UC framework if the environment cannot distinguish between real-world and ideal-world execution. Formally, protocol π UC-realizes $\mathcal{F}$ if for every probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for every PPT environment $\mathcal{Z}$

$$\{\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}}(1^\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} = \{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(1^\lambda, z)\}_{\lambda\in\mathbb{N},z\in\{0,1\}^*} .$$

C Anonymous Credentials and Blind Signing

Our online protocol defined in Section 4.2 describes a threshold variant of the BBS+ signature scheme. Since anonymous credentials are one prominent application of BBS+ signatures, we elaborate on this application in the following.

BBS+ signatures can be used to design anonymous credential schemes as follows. To receive a credential, a client sends a signing request to the servers in the form of a message array, which contains its public and private credential information. Public parts of the credentials are sent in clear, while private information is blinded. The client can add zero-knowledge proofs that blinded messages satisfy some predicate. These proofs enable the issuing servers to enforce a signing policy even though they blindly sign parts of the messages. Given a credential, clients can prove in zero-knowledge that their credential fulfills certain predicates without leaking their signature.

Our scheme must be extended by a blind-signing property to realize the described blueprint. Precisely, we require a property called *partially blind* signatures [1]. This property prevents the issuer from learning private information about the message to be signed.

To transform our scheme into a partially blind signature scheme, we follow the approach of [5]. Let $\{m_\ell\}_{\ell\in[k]}$ be the set of messages representing the client's credential information. Without loss of generality, we assume that m_k is the public part. In order to blind its messages, the client computes a Pedersen commitment [61] on the private messages: $C = h_0^{s'} \cdot \prod_{\ell\in[k-1]} h_\ell^{m_\ell}$ for a random s' and a zero-knowledge proof π that C is well-formed, i.e., that the client knows $(s', \{m_\ell\}_{\ell\in[k-1]})$. The client sends $(\mathcal{T}, C, \pi, m_k)$ and potential zero-knowledge proofs for signing policy enforcement to the servers. Each server P_i for $i \in \mathcal{T}$ replies with $(A_i = (g_1 \cdot C \cdot h_k^{m_k})^{a_i} \cdot h_0^{\alpha_i}, \delta_i, e_i, s_i)$. The client computes e, s, and A as before but outputs signature $(A, e, s^* = s' + s)$ which yields a valid signature.

As the blinding mechanism and the resulting signatures are equivalent in the non-threshold BBS+ setting, we can use existing zero-knowledge proofs for policy enforcement and credential usage from the non-threshold setting [5, 23, 67].

D Reusable PCG Constructions

In this section, we present constructions of reusable PCGs for VOLE and OLE correlations according to the definitions provided in Section 3 together with the required building blocks and security assumptions. The constructions are derived from the one of [19].

Notation Let R be a ring. For two column vectors $\mathbf{u} = (u_1, \dots, u_t) \in R^t$ and $\mathbf{v} = (v_1, \dots, v_t) \in R^t$, we define the *outer sum* $\mathbf{u} \boxplus \mathbf{v}$ be the vector $(u_i + v_j)_{i,j \in [t]} \in R^{t^2}$. Similar, we define the *outer product* (or tensor product) $\mathbf{u} \otimes \mathbf{v}$ to be $(u_i \cdot v_j)_{i,j \in [t]} \in R^{t^2}$. The *inner product* of two t-size vectors $\langle \mathbf{u}, \mathbf{v} \rangle$ is defined as $\left(\sum_{i \in [t]} u_i \cdot v_i\right) \in R$.

Ring Module LPN Assumption The following definition of the Module Ring LPN assumption introduced by [19] is taken almost verbatim from the original [19, Definition 3.2] but adapted to our notation.

Definition 3. *Module-LPN*

Let $c \geq 2$ be an integer, let $R = \mathbb{Z}_p / F(X)$ for a prime p and degree-N polynomial $F(X) \in \mathbb{Z}_p[X]$ and let $\tau \in \mathbb{N}$ be an integer. Further, let $\mathcal{HW}_{R,\tau}$ denote the distribution of "sparse polynomials" over R obtained by sampling τ noise positions $\alpha \leftarrow [N]^\tau$ and τ payloads $\beta \leftarrow (\mathbb{Z}_p^*)^\tau$ uniformly at random and outputting $e(X) := \sum_{i \in [\tau]} \beta[i] \cdot X^{\alpha[i]-1}$. Then, for $R = R(\lambda), m = m(\lambda), \tau = \tau(\lambda)$, we say the R^c-$\mathsf{LPN}_{R,m,\tau}$ problem is hard if for every nonuniform polynomial-time distinguisher $\mathcal{A}$, it holds that

$$|\Pr[\mathcal{A}(\{(\mathbf{a}^{(i)}, \langle \mathbf{a}^{(i)}, \mathbf{e} \rangle + f^{(i)})\}_{i \in [m]}) = 1]$$
$$-\Pr[\mathcal{A}(\{(\mathbf{a}^{(i)}, u^{(i)})\}_{i \in [m]}) = 1]| \leq \mathsf{negl}(\lambda)$$

where the probabilities are taken over $\mathbf{a}^{(1)}, \dots, \mathbf{a}^{(m)} \leftarrow R^{c-1}$, $u^{(1)}, \dots, u^{(m)} \leftarrow R$, $\mathbf{e} \leftarrow \mathcal{HW}_{R,\tau}^{c-1}$, $f^{(1)}, \dots, f^{(m)} \leftarrow \mathcal{HW}_{R,\tau}$.

Distributed Sum of Point Functions (DSPF) We use distributed sum of functions. The definition is taken partially verbatim from [19, 2] but adapted to our notation.

Definition 4 (Distributed Sum of Point Functions). *Let $\mathbb{G}$ be an Abelian group, N, τ be positive integers, $f_{\alpha,\beta} : [N] \to \mathbb{G}$ be a sum of τ point functions, parametrized for $\alpha \in [N]^\tau$ and $\beta \in \mathbb{G}^\tau$, such that $f_{\alpha,\beta}(x) = 0 + \sum_{(i \in [\tau] \ s.t. \ \alpha[i]=x)} \beta[i]$. A 2-party distributed sum of point functions (DSPF) with domain $[N]$, codomain $\mathbb{G}$, and weight τ is a pair of PPT algorithms* $(\mathsf{DSPF.Gen}, \mathsf{DSPF.Eval})$ *with the following syntax.*

- $\mathsf{DSPF.Gen}$ *takes as input the security parameter 1^λ and a description of the sum of point functions $f_{\alpha,\beta}$, specifically, the special positions $\alpha \in [N]^\tau$ and the non-zero elements $\beta \in \mathbb{G}^\tau$. The output is two keys (K_0, K_1).*
- $\mathsf{DSPF.Eval}$ *takes as input a DPF key K_σ, index $\sigma \in \{0,1\}$ and a value $x \in [N]$, outputting an additive share v_σ of $f_{\alpha,\beta}(x)$.*

A DSPF should satisfy the following properties:

- ***Correctness.*** *For every set of special positions $\alpha \in [N]^\tau$, set of non-zero elements $\beta \in \mathbb{G}^\tau$ and element $x \in [N]$, we have that*

$$\Pr[v_0 + v_1 = f_{\alpha,\beta}(x) | (K_0, K_1) \leftarrow \mathsf{DSPF.Gen}(1^\lambda, \alpha, \beta),$$
$$v_\sigma \leftarrow \mathsf{DSPF.Eval}(K_\sigma, \sigma, x) \text{ for } \sigma \in \{0,1\}] = 1$$

– ***Security.*** *There exists a PPT simulator $\mathcal{S}$ such that, for every corrupted party $\sigma \in \{0,1\}$, set of special positions $\alpha \in [N]^\tau$ and set of non-zero elements $\beta \in \mathbb{G}^\tau$, the output of $\mathcal{S}(1^\lambda, \sigma)$ is computationally indistinguishable from*

$$\{K_\sigma | (K_0, K_1) \leftarrow \mathsf{DSPF.Gen}(1^\lambda, \alpha, \beta)\}$$

We denote the execution of $\mathsf{DSPF.Eval}(K_\sigma, \sigma, x)$ for every $x \in [N]$, i.e. the evaluation over the whole domain $[N]$, by $\mathsf{DSPF.FullEval}(K_\sigma, \sigma)$.

PCG constructions. The OLE construction is derived from [19, Fig. 1]. However, we extend it by the reusability feature by deriving the sparse polynomials normally sampled in $\mathsf{PCG.Gen}$ by applying a random oracle on seeds provided as input to the programmable key generation $\mathsf{PCG.Gen_p}$.

Construction 3: Reusable PCG for $\mathcal{Y}^R_{\mathsf{OLE}}$

Let λ be the security parameter, $\tau = \tau(\lambda)$ be the noise weight, $c \geq 2$ the compression factor, $p = p(\lambda)$ a modulus, $N = N(\lambda)$ a degree, and $R_p = \mathbb{Z}_p[X]/F(X)$ be a ring for a degree-N $F(X) \in \mathbb{Z}_p[X]$. Further, let ($\mathsf{DSPF.Gen}, \mathsf{DSPF.Eval}$) be a FSS scheme for sums of τ^2-point functions with domain $[2N]$ and range $\mathbb{Z}_p$. Finally, let $\mathcal{H} : \{0,1\}^\lambda \rightarrow ([N]^\tau \times (\mathbb{Z}_p^*)^\tau)^c$ be a random oracle.
Correlation: The target correlation $\mathcal{Y}^R_{\mathsf{OLE}}$ over ring R_p is defined as

$$\mathsf{mk} = (\rho_0, \rho_1) \leftarrow \mathsf{Setup}^R_{\mathsf{OLE}}(1^\lambda, \rho_0, \rho_1)$$
$$((x_0, z_0), (x_1, z_1)) \leftarrow \mathcal{Y}^R_{\mathsf{OLE}}(1^\lambda, \mathsf{mk}) \text{ such that}$$
$$x_0 = F_0(\rho_0), x_1 = F_1(\rho_1), z_0 \overset{\$}{\leftarrow} R_p, z_1 = x_0 \cdot x_1 - z_0$$
$$(x_\sigma, x_0 \cdot x_1 - z_\sigma) \leftarrow \mathsf{RSample}^R_{\mathsf{OLE}}(1^\lambda, \mathsf{mk}, \sigma, (x_\sigma, z_\sigma)) \text{ where}$$
$$x_0 = F_0(\rho_0), x_1 = F_1(\rho_1)$$

with F_0 and F_1 being PRGs. As proposed by [19], R_p can be constructed to be isomorphic to N copies of $\mathbb{Z}_p$. This allows the direct transformation of one OLE over R_p into N independent OLEs over $\mathbb{Z}_p$.
Public Input: Random $R^c - \mathsf{LPN}$ polynomials $a_2, \dots a_c \in R_p$, defining the vector $\mathbf{a} = (1, a_2, \dots, a_c)$.
$\mathsf{PCG.Gen_p}(1^\lambda, \rho_0, \rho_1)$**:**

1. Compute $\{(\alpha^{\mathbf{i}}_\sigma, \beta^i_\sigma)\}_{i \in [c]} \leftarrow \mathcal{H}(\rho_\sigma)$ for $\sigma \in \{0,1\}$ where each $\alpha^i_\sigma \in [N]^\tau$ and each $\beta^i_\sigma \in (\mathbb{Z}_p^*)^\tau$.
2. For $i, j \in [c]$, sample FSS keys $(K_0^{(i,j)}, K_1^{(i,j)}) \overset{\$}{\leftarrow} \mathsf{DSPF.Gen}(1^\lambda, \alpha^i_0 \boxplus \alpha^j_1, \beta^i_0 \otimes \beta^j_1)$.
3. For $\sigma \in \{0,1\}$, define $\mathsf{k}_\sigma = (\{(\alpha^{\mathbf{i}}_\sigma, \beta^i_\sigma)\}_{i \in [c]}, \{K_\sigma^{(i,j)}\}_{i,j \in [c]})$.
4. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

$\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)$**:**

5. Parse k_σ as $(\{(\alpha^{\mathbf{i}}_\sigma, \beta^i_\sigma)\}_{i \in [c]}, \{K_\sigma^{(i,j)}\}_{i,j \in [c]})$.
6. For $i \in [c]$, define (over $\mathbb{Z}_p$) the degree $< N$ polynomial:
$$e^i_\sigma(X) = \sum_{k \in [\tau]} \beta^i_\sigma[k] \cdot X^{\alpha^i_\sigma[k]}$$
and compose all e^i_σ (for $i \in [c]$) to a length-c vector $\mathbf{e}_\sigma$.
7. For $i, j \in [c]$, compute $u_\sigma^{i+c(j-1)} \leftarrow \mathsf{DSPF.FullEval}(\sigma, K_\sigma^{(i,j)})$ and view this as a degree $< 2N$ polynomial. Compose all u^i_σ (for $i \in [c^2]$) to a length-c^2 vector $\mathbf{v}_\sigma$ mod $F(X)$.
8. Compute $x_\sigma = \langle \mathbf{a}, \mathbf{e}_\sigma \rangle \mod F(X)$ and $z_\sigma = \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{v}_\sigma \rangle \mod F(X)$.
9. Output (x_σ, z_σ).

From the previous construction, we derive a VOLE construction in a straight-forward way.

Construction 4: Reusable PCG for $\mathcal{Y}^R_{\mathsf{VOLE}}$

Let λ be the security parameter, $\tau = \tau(\lambda)$ be the noise weight, $c \geq 2$ the compression factor, $p = p(\lambda)$ a modulus, $N = N(\lambda)$ a degree, and $R_p = \mathbb{Z}_p[X]/F(X)$ be a ring for a degree-N $F(X) \in \mathbb{Z}_p[X]$. Further, let $(\mathsf{DSPF.Gen}, \mathsf{DSPF.Eval})$ be a FSS scheme for sums of τ point functions with domain $[N]$ and range $\mathbb{Z}_p$. Finally, let $\mathcal{H} : \{0,1\}^\lambda \to ([N]^\tau \times (\mathbb{Z}_p^*)^\tau)^c$ be a random oracle.
Correlation: The target correlation $\mathcal{Y}^R_{\mathsf{VOLE}}$ over ring R_p is defined as

$$\mathsf{mk} = (\rho, x) \leftarrow \mathsf{Setup}^R_{\mathsf{OLE}}(1^\lambda, \rho, x)$$
$$((y, z_0), (x, z_1)) \leftarrow \mathcal{Y}^R_{\mathsf{OLE}}(1^\lambda, \mathsf{mk}) \text{ such that}$$
$$y = F(\rho), z_0 \xleftarrow{\$} R_p, z_1 = x \cdot y - z_0$$
$$(x, x \cdot F(\rho) - z_0) \leftarrow \mathsf{RSample}^R_{\mathsf{VOLE}}(1^\lambda, \mathsf{mk}, 0, (F(\rho), z_0))$$
$$(F(\rho), x \cdot y - z_1) \leftarrow \mathsf{RSample}^R_{\mathsf{VOLE}}(1^\lambda, \mathsf{mk}, 1, (x, z_1))$$

with F being a PRG. As proposed by [19], R_p can be constructed to be isomorphic to N copies of $\mathbb{Z}_p$. This allows the direct transformation of one VOLE over R_p into N independent VOLEs over $\mathbb{Z}_p$.
Public Input: Random $R^c - \mathsf{LPN}$ polynomials $a_2, \ldots a_c \in R_p$, defining the vector $\mathbf{a} = (1, a_2, \ldots, a_c)$.
$\mathsf{PCG.Gen_p}(1^\lambda, \rho_0, \rho_1)$**:**

1. Parse ρ_1 as x and compute $\{(\alpha^i, \beta^i)\}_{i \in [c]} \leftarrow \mathcal{H}(\rho_0)$ where $x \in \mathbb{Z}_p^*$, each $\alpha^i \in [N]^\tau$ and each $\beta^i \in (\mathbb{Z}_p^*)^\tau$.
2. For $i \in [c]$, sample FSS keys $(K_0^i, K_1^j) \xleftarrow{\$} \mathsf{DSPF.Gen}(1^\lambda, \alpha^i, x \cdot \beta^i)$.
3. For $\sigma \in \{0,1\}$, define $\mathsf{k}_\sigma = (\rho_\sigma, \{K_\sigma^i\}_{i \in [c]})$.
4. Output $(\mathsf{k}_0, \mathsf{k}_1)$.

$\mathsf{PCG.Expand}(\sigma, \mathsf{k}_\sigma)$**:**

5. If $\sigma = 0$, parse k_0 as $(\rho_0, \{K_0^i\}_{i \in [c]})$ and compute $\{(\alpha^i, \beta^i)\}_{i \in [c]} \leftarrow \mathcal{H}(\rho_0)$ where each $\alpha^i \in [N]^\tau$ and each $\beta^i \in (\mathbb{Z}_p^*)^\tau$. Then, for $i \in [c]$, define (over $\mathbb{Z}_p$) the degree $< N$ polynomial:
$$e^i(X) = \sum_{k \in [\tau]} \beta^i[k] \cdot X^{\alpha^i[k]}$$
and compose all e^i (for $i \in [c]$) to a length-c vector $\mathbf{e}$.
6. If $\sigma = 1$, parse k_1 as $(x, \{K_1^i\}_{i \in [c]})$.
7. For $i \in [c]$, compute $u_\sigma^i \leftarrow \mathsf{DSPF.FullEval}(\sigma, K_\sigma^i)$ and view the result as a degree $< N$ polynomial. Compose all u_σ^i (for $i \in [c]$) to a length-c vector $\mathbf{v}_\sigma \mod F(X)$.
8. Compute $z_\sigma = \langle \mathbf{a}, \mathbf{v}_\sigma \rangle \mod F(X)$.
9. If
   - $\sigma = 0$, compute $y = \langle \mathbf{a}, \mathbf{e} \rangle \mod F(X)$ and output (y, z_0)
   - $\sigma = 1$, output (x, z_1).

Security. We state the following Theorems:

Theorem 3. *Assume the R^c-$\mathsf{LPN}_{R_p,1,\tau}$ assumption holds and that* DSPF *is a secure instantiation of a distributed sum of point functions. Then, Construction 3 is a secure reusable PCG for OLE correlations over R_p in the random oracle model.*

Theorem 4. *Assume the R^c-$\mathsf{LPN}_{R_p,1,\tau}$ assumption holds and that* DSPF *is a secure instantiation of a distributed sum of point functions. Then, Construction 4 is a secure reusable PCG for VOLE correlations over R_p in the random oracle model.*

In the following, we provide a proof sketch for Theorem 3. A proof sketch for Theorem 4 follows in a straight-forward way.

Proof. To show that Construction 3 is a secure reusable PCG, we need to show programmability, pseudorandom $\mathcal{Y}$-correlated outputs, security and key indistinguishability.

Programmability can be shown, by defining ϕ_σ as a function, that first computes $\{(\alpha^{\mathbf{i}}_\sigma, \beta^i_\sigma)\}_{i\in[c]} \leftarrow \mathcal{H}(\rho_\sigma)$, expands these to $\mathbf{e}_\sigma \in R_p^c$ as done in the $\mathsf{PCG.Expand}$ algorithm, and then outputs $\langle \mathbf{a}, \mathbf{e}_\sigma\rangle$.

Pseudorandom $\mathcal{Y}$-correlated outputs can be shown via a sequence of games. First, we replace the PRG F_σ in $\mathcal{Y}$ by ϕ_σ. As the random oracle ensures that the secrets e^*_* are sampled uniformly at random, indistinguishability can be shown via a reduction to the R^c-$\mathsf{LPN}_{R_p,1,\tau}$ assumption. Next, we skip the DSPF key generation and full evaluation during the expansion. Instead, we directly sample $z_0 \in_R R_p$ and define $z_1 = x_0 \cdot x_1 - z_0$. Here, indistinguishability can be shown analogously to the correctness proof in [19]. Note that in the previous game for every $i, j \in [c]$, it holds that

$$e^i_0(X) \cdot e^j_1(X) = \sum_{k,l\in[\tau]} \beta^i_0[k] \cdot \beta^j_1[l] \cdot X^{\alpha^i_0[k]\cdot\alpha^j_1[l]}.$$

Therefore, parties can obtain an additive sharing of this product by fully evaluating the (i, j)-th DSPF instance. It follows that $u_0^{i+c(j-1)} + u_1^{i+c(j-1)} = e^i_0(X) \cdot e^j_1(X)$, and hence, $\mathbf{v} = \mathbf{e}_0 \otimes \mathbf{e}_1$. This observation yields the following relation of the outputs:

$$\begin{aligned} z_0 + z_1 &= \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{v}_0 + \mathbf{v}_1\rangle = \langle \mathbf{a} \otimes \mathbf{a}, \mathbf{e}_0 + \mathbf{e}_1\rangle \\ &= \langle \mathbf{a}, \mathbf{e}_0\rangle \cdot \langle \mathbf{a}, \mathbf{e}_1\rangle = x_0 \cdot x_1 \end{aligned}$$

As the correlation of (x_0, x_1, z_0, z_1) is the same in both games, the computation of x_0 and x_1 remains untouched, and the DSPF implies that each z_σ is individually pseudorandom, both games are computationally indistinguishable. In the resulting game, the challenger executes the exact same steps independent of the coin b. Therefore, it follows that any adversary wins the final game with probability exactly $\frac{1}{2}$ which implies that any adversary wins the original security game with probability at most $\frac{1}{2} + \mathsf{negl}$.

As $\mathsf{RSample}^R_{\mathsf{OLE}}$ executes the same steps as the forward sampling $\mathcal{Y}^R_{\mathsf{OLE}}$, security can be shown analogously to the pseudorandom $\mathcal{Y}$-correlated outputs property.

Key indistinguishability follows from the security property of the DSPF scheme via a sequence of game hops. We replace one by one the DSPF-keys in k_σ with ones produced by the DSPF-simulator. Indistinguishability between games can be proven via reductions to the security property of the DSPF scheme. Finally, we remove in one more game the PCG key generation and the assignment of $\rho_{1-\sigma}$ as both steps become redundant. The final game is completely independent of the choice of b such that the success probability of $\mathcal{A}$ is exactly $\frac{1}{2}$ which shows that the success probability of $\mathcal{A}$ in the initial game is at most $\frac{1}{2} + \mathsf{negl}(\lambda)$.

E Reusable Pseudorandom Correlation Function

On a high level, a pseudorandom correlation function (PCF) allows two parties to generate a large amount of correlated randomness from short seeds. PCF extends the notion of a pseudorandom correlation generator (PCG) in a similar way as a pseudorandom function extends a pseudorandom generator. While a PCG generates a large batch of correlated randomness during one-time expansion, a PCF allows the creation of correlation samples on the fly.

A PCF consists of two algorithms, Gen and Eval. The Gen algorithm computes a pair of short keys distributed to two parties. Then, each party can locally evaluate the Eval algorithm using its key and public input to generate an output of the target correlation. One example of such a correlation is the oblivious linear evaluation (OLE) correlation, defined by a pair of random values (y_0, y_1) where $y_0 = (a, u)$ and $y_1 = (s, v)$ such that $v = as + u$. Other meaningful correlations are oblivious transfer (OT) and multiplication triples.

PCFs are helpful in two- and multi-party protocols, where parties first set up correlated randomness and then use this data to speed up the computation [41, 2, 51].

This section presents our definition of reusable PCFs, extending the definition of programmable PCFs from [18]. Furthermore, we state constructions of reusable PCFs and argue why they satisfy our new definition in Appendix F.

Our modifications and extensions of the definition [18] reflect the challenges we faced when using PCFs as black-box primitives in our threshold BBS+ protocol. We present our definition and highlight these challenges and changes in the following.

E.1 Definition

Similar to PCGs, PCFs realize a target correlation $\mathcal{Y}$. While PCFs output single correlation outputs instead of a bunch of correlation as PCGs, we need to slightly adapt the definition of a target correlation. We emphasize the modification in the following.

We formally define a *target correlation* as a tuple of probabilistic algorithms $(\mathsf{Setup}, \mathcal{Y})$, where Setup takes two inputs and creates a master key mk. These inputs enable fixing parts of the correlation, e.g., the fixed value s. Algorithm $\mathcal{Y}$ uses the master key and an index i to sample correlation outputs. The index i helps to sample the same value if one of the Setup inputs is identical for multiple invocations. The input i is not necessary for correlations for PCGs since the output of PCG expansion is a bunch of correlation. For PCFs, the output of the evaluation is a single correlation tuple. Thus, we need the index i to sample the same value if one of the Setup inputs is identical for multiple PCF invocations.

Definition 5 (Reverse-sampleable and indexable target correlation with setup). *Let $\ell_0(\lambda), \ell_1(\lambda) \leq \mathsf{poly}(\lambda)$ be output length functions. Let $(\mathsf{Setup}, \mathcal{Y})$ be a tuple of probabilistic algorithms, such that Setup on input 1^λ and two parameters ρ_0, ρ_1 returns a master key mk; algorithm $\mathcal{Y}$ on input 1^λ, mk, and index i returns a pair of outputs $(y_0^{(i)}, y_1^{(i)}) \in \{0,1\}^{\ell_0(\lambda)} \times \{0,1\}^{\ell_1(\lambda)}$.*

We say that the tuple $(\mathsf{Setup}, \mathcal{Y})$ defines a reverse-sampleable and indexable target correlation with setup *if there exists a probabilistic polynomial time algorithm $\mathsf{RSample}$ that takes as input $1^\lambda, \mathsf{mk}, \sigma \in \{0,1\}, y_\sigma^{(i)} \in \{0,1\}^{\ell_\sigma(\lambda)}$ and i, and outputs $y_{1-\sigma}^{(i)} \in \{0,1\}^{\ell_{1-\sigma}(\lambda)}$, such that for all $\sigma \in \{0,1\}$, for all $\mathsf{mk}, \mathsf{mk}'$ in the range of Setup for arbitrary but fixed input ρ_σ, and all $i \in \{0,1\}^*$ the following distributions are statistically close:*

$$\{(y_0^{(i)}, y_1^{(i)}) | (y_0^{(i)}, y_1^{(i)}) \xleftarrow{\$} \mathcal{Y}(1^\lambda, \mathsf{mk}, i)\}$$
$$\{(y_0^{(i)}, y_1^{(i)}) | (y_0'^{(i)}, y_1'^{(i)}) \xleftarrow{\$} \mathcal{Y}(1^\lambda, \mathsf{mk}', i),$$
$$y_\sigma^{(i)} \leftarrow y_\sigma'^{(i)}, y_{1-\sigma}^{(i)} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{mk}, \sigma, y_\sigma, i)\}.$$

Given the definition of a reverse-sampleable and indexable correlation with setup, we define our primitive called *strong reusable PCF* (srPCF). Our definition builds on the definition of a strong PCF of Boyle et al. [18] and extends it by a reusability feature. Note that [18] presents a separate definition of this reusability feature for PCFs, but this property also affects the other properties of a PCF. Therefore, we merge these definitions. Additionally, the reusability definition of Boyle et al. works only for the semi-honest setting, while our definition covers malicious adversaries. The crucial point to cover malicious adversaries is to allow the corrupted party to choose an arbitrary value as its input to the key generation. Our definitions give this power to the adversary, while the definitions of Boyle et al. use randomly chosen inputs.

A PCF must fulfill two properties. First, the pseudorandomness property intuitively states that the joint outputs of the Eval algorithm are computationally indistinguishable from outputs of the correlation $\mathcal{Y}$. Second, the security property intuitively guarantees that the PCF output of party $P_{1-\sigma}$ is indistinguishable from a reverse-sampled value. Indistinguishability holds even if the adversary corrupts party P_σ and learns its key. Hence, this property provides security against an insider.

Similarly to the notions of weak and strong PRFs, there exist the notions of *weak* and *strong* PCFs. For a weak PCF, we consider the Eval algorithm to be executed on randomly chosen inputs, while for a strong PCF, we consider arbitrarily chosen inputs. Boyle et al. [18] showed a generic transformation from a weak to a strong PCF using a hash function modeled as a programmable random oracle. In Appendix F, we present constructions for weak srPCFs, which then yield strong srPCFs based on the transformation of Boyle et al.

A PCF needs to meet two additional requirements to satisfy the reusability features. First, an adversary cannot learn any information about the other party's input used for the key generation from its own key. This is modeled by the key indistinguishability property and the corresponding game in Figure 10. In the game, the challenger samples two random values and uses one for the key generation. Then, given the corrupted party's key and the random values, the adversary has to identify which of the two random value was used. Second, two efficiently computable functions must exist to compute the reusable parts of the correlation from the setup input and the public evaluation input. Formally, we state the definition of a strong reusable PCF next.

Definition 6 (Strong reusable pseudorandom correlation function (srPCF)). *Let* $(\mathsf{Setup}, \mathcal{Y})$ *be a reverse-sampleable and indexable correlation with setup which has output length functions* $\ell_0(\lambda), \ell_1(\lambda)$*, and let* $\lambda \leq \eta(\lambda) \leq \mathsf{poly}(\lambda)$ *be an input length function. Let* $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *be a pair of algorithms with the following syntax:*

– $\mathsf{PCF.Gen}(1^\lambda, \rho_0, \rho_1)$ *is a probabilistic polynomial-time algorithm that on input the security parameter* 1^λ *and reusable inputs* ρ_0, ρ_1 *outputs a pair of keys* $(\mathsf{k}_0, \mathsf{k}_1)$.
– $\mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$ *is a deterministic polynomial-time algorithm that on input* $\sigma \in \{0,1\}$*, key* k_σ *and input value* $x \in \{0,1\}^{\eta(\lambda)}$ *outputs a value* $y_\sigma \in \{0,1\}^{\ell_\sigma(\lambda)}$.

We say $(\mathsf{PCF.Gen}, \mathsf{PCF.Eval})$ *is a strong reusable pseudorandom correlation function (srPCF) for* $(\mathsf{Setup}, \mathcal{Y})$*, if the following conditions hold:*

– ***Strong pseudorandom*** $\mathcal{Y}$***-correlated outputs.*** *For every non-uniform adversary* $\mathcal{A}$ *of size* $\mathsf{poly}(\lambda)$ *asking at most* $\mathsf{poly}(\lambda)$ *queries to the oracle* $\mathcal{O}_b(\cdot)$*, it holds*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{s\text{-}pr}}_{\mathcal{A}}(\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$$

for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{s\text{-}pr}}_{\mathcal{A}}(\lambda)$ *is as defined in Figure 8.*
– ***Strong security.*** *For each* $\sigma \in \{0,1\}$ *and non-uniform adversary* $\mathcal{A}$ *of size* $\mathsf{poly}(\lambda)$ *asking at most* $\mathsf{poly}(\lambda)$ *queries to oracle* $\mathcal{O}_b(\cdot)$*, it holds*

$$\left|\Pr[\mathsf{Exp}^{\mathsf{s\text{-}sec}}_{\mathcal{A},\sigma}(\lambda) = 1] - \frac{1}{2}\right| \leq \mathsf{negl}(\lambda)$$

for all sufficiently large λ*, where* $\mathsf{Exp}^{\mathsf{s\text{-}sec}}_{\mathcal{A},\sigma}(\lambda)$ *is as defined in Figure 9.*

```
$\mathsf{Exp}^{\text{s-pr}}_{\mathcal{A}}(\lambda)$ :
$(\rho_0, \rho_1) \leftarrow \mathcal{A}_0(1^\lambda)$
$\mathsf{mk} \leftarrow \mathsf{Setup}(1^\lambda, \rho_0, \rho_1)$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda, \rho_0, \rho_1)$
$\mathcal{Q} = \emptyset$
$b \xleftarrow{\$} \{0,1\}$
$b' \leftarrow \mathcal{A}_1^{\mathcal{O}_b(\cdot)}(1^\lambda)$
if $b = b'$ return $1$
else return $0$

$\mathcal{O}_0(x)$ :
if $(x, y_0, y_1) \in \mathcal{Q}$ :
  return $(y_0, y_1)$
else :
  $(y_0, y_1) \leftarrow \mathcal{Y}(1^\lambda, \mathsf{mk}, x)$
  $\mathcal{Q} = \mathcal{Q} \cup \{(x, y_0, y_1)\}$
  return $(y_0, y_1)$

$\mathcal{O}_1(x)$ :
for $\sigma \in \{0,1\}$ :
  $y_\sigma \leftarrow \mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$
return $(y_0, y_1)$
```

Fig. 8: Strong pseudorandom $\mathcal{Y}$-correlated outputs of a PCF.

– ***Programmability.*** *There exist public efficiently computable functions f_0, f_1, such that for all $x \in \{0,1\}^{\eta(\lambda)}$ and all $\rho_0, \rho_1 \in \{0,1\}^*$*

$$\Pr\left[\begin{array}{l} (\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda, \rho_0, \rho_1) \\ (a, c) \leftarrow \mathsf{PCF.Eval}(0, \mathsf{k}_0, x), \\ (b, d) \leftarrow \mathsf{PCF.Eval}(1, \mathsf{k}_1, x) \end{array} : \begin{array}{c} a = f_0(\rho_0, x) \\ b = f_1(\rho_1, x) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

– ***Key indistinguishability.*** *For any $\sigma \in \{0,1\}$ and non-uniform adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, it holds*

$$\Pr[\mathsf{Exp}^{\text{key-ind}}_{\mathcal{A},\sigma}(\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$$

for all sufficiently large λ, where $\mathsf{Exp}^{\text{key-ind}}_{\mathcal{A},\sigma}$ is as defined in Figure 10.

```
$\mathsf{Exp}^{\text{s-sec}}_{\mathcal{A},\sigma}(\lambda)$ :
$(\rho_0, \rho_1) \leftarrow \mathcal{A}_0(1^\lambda)$
$\mathsf{mk} \leftarrow \mathsf{Setup}(1^\lambda, \rho_0, \rho_1)$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda, \rho_0, \rho_1)$
$b \xleftarrow{\$} \{0,1\}$
$b' \leftarrow \mathcal{A}_1^{\mathcal{O}_b(\cdot)}(1^\lambda, \sigma, \mathsf{k}_\sigma)$
if $b = b'$ return $1$
else return $0$

$\mathcal{O}_0(x)$ :
$y_{1-\sigma} \leftarrow \mathsf{PCF.Eval}(1-\sigma, \mathsf{k}_{1-\sigma}, x)$
return $y_{1-\sigma}$

$\mathcal{O}_1(x)$ :
$y_\sigma \leftarrow \mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$
$y_{1-\sigma} \leftarrow \mathsf{RSample}(1^\lambda, \mathsf{mk}, \sigma, y_\sigma, x)$
return $y_{1-\sigma}$
```

Fig. 9: Strong security of a PCF.

```
$\underline{\mathsf{Exp}^{\mathsf{key\text{-}ind}}_{\mathcal{A},\sigma}(\lambda):}$
$b \xleftarrow{\$} \{0,1\}$
$\rho^{(0)}_{1-\sigma}, \rho^{(1)}_{1-\sigma} \xleftarrow{\$} \{0,1\}^*$
$\rho_{1-\sigma} \leftarrow \rho^{(b)}_{1-\sigma}$
$\rho_\sigma \leftarrow \mathcal{A}_0(1^\lambda)$
$(\mathsf{k}_0, \mathsf{k}_1) \leftarrow \mathsf{PCF.Gen}(1^\lambda, \rho_0, \rho_1)$
$b' \leftarrow \mathcal{A}_1(1^\lambda, \mathsf{k}_\sigma, \rho^{(0)}_{1-\sigma}, \rho^{(1)}_{1-\sigma})$
**if** $b' = b$ **return** 1
**else return** 0
```

Fig. 10: Key Indistinguishability of a reusable PCF.

E.2 Correlations

Here, we state the correlations required for our PCF-based precomputation protocol (cf. Appendix G.2). As these correlations differ slightly from the correlations required by our PCG-based offline phase (cf. Section 5), we state them in the following for completeness.

Our OLE correlation over ring R is given by $c_1 = ab + c_0$, where $a, b, c_0, c_1 \in R$. Moreover, we require a and b being computed by a weak pseudorandom function (PRF). Formally, we define the reverse-sampleable and indexable target correlation with setup $(\mathsf{Setup}_{\mathsf{OLE}}, \mathcal{Y}_{\mathsf{OLE}})$ over ring R as

$$\begin{aligned} (k, k') &\leftarrow \mathsf{Setup}_{\mathsf{OLE}}(1^\lambda, k, k'), \\ ((F_k(i), u), (F_{k'}(i), v)) &\leftarrow \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (k, k'), i) \quad \text{such that} \\ v &= F_k(i) \cdot F_{k'}(i) + u, \end{aligned} \tag{5}$$

where $u \xleftarrow{\$} R, u \in R$ and F being a (PRF) with key k, k'. Note that while the Setup algorithm for our OLE and VOLE correlation essentially is the identity function, the algorithm might be more complex for other correlations. The reverse-sampling algorithm is defined such that $(F_{k'}(i), F_k(i) \cdot F_{k'}(i) + u) \leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (k, k'), 0, (F_k(i), u), i)$ and $(F_k(i), v - F_k(i) \cdot F_{k'}(i)) \leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (k, k'), 1, (F_{k'}(i), v), i)$.

Our VOLE correlation is the same as OLE but the value b is fixed over multiple correlation samples, i.e., $\vec{c_1} = \vec{a}b + \vec{c_0}$, where each correlation sample contains one component of the vectors. We formally define the reverse-sampleable and indexable target correlation with setup $(\mathsf{Setup}_{\mathsf{VOLE}}, \mathcal{Y}_{\mathsf{VOLE}})$ over ring R as

$$\begin{aligned} (k, b) &\leftarrow \mathsf{Setup}_{\mathsf{VOLE}}(1^\lambda, k, b), \\ ((F_k(i), u), (b, v)) &\leftarrow \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (k, b), i) \quad \text{such that} \\ v &= F_k(i) \cdot b + u, \end{aligned} \tag{6}$$

where $u \xleftarrow{\$} R, b, v \in R$ and F being a weak pseudorandom function (PRF) with key k. Note that b is fixed over all correlation samples, while u and v are not. The reverse-sampling algorithm is defined such that

$$\begin{aligned} (b, F_k(i) \cdot b + u) &\leftarrow \mathsf{RSample}_{\mathsf{VOLE}}(1^\lambda, (k, b), 0, (F_k(i), u), i) \text{ and} \\ (F_k(i), v - F_k(i) \cdot b) &\leftarrow \mathsf{RSample}_{\mathsf{VOLE}}(1^\lambda, (k, b), 1, (b, v), i). \end{aligned}$$

We state PCF constructions realizing these definitions of OLE and VOLE correlations in Appendix F. The VOLE PCF construction is taken from [18], and the OLE PCF follows a straightforward adaptation of the VOLE PCF.

F Reusable PCF Constructions

This sections presents construction of reusable PCFs for VOLE and OLE correlations as defined in Section 3.2. We first present the reusable PCF for VOLE and then for OLE.

The VOLE construction heavily builds on the constructions of [18], which provides only weak PCF. However, Boyle et al. presented a generic transformation from weak to strong PCF using a programmable random oracle. This transformation is also straightforwardly applicable to reusable PCFs. Therefore, we state a weak reusable PCF in the following and emphasize that this construction can be extended to a strong reusable PCF in the programmable random oracle model.

The following construction is taken from [18, Fig. 22]. It builds on a weak PRF F and a function secret sharing for the multiplication of F with a scalar.

Construction 5: Reusable PCF for $\mathcal{Y}_{\mathsf{VOLE}}$

Let $\mathcal{F} = \{F_k : \{0,1\}^\eta \to R\}_{k\in\{0,1\}^\lambda}$ be a weak PRF and $\mathsf{FFS} = (\mathsf{FFS.Gen}, \mathsf{FFS.Eval})$ an FSS scheme for $\{c \cdot F_k\}_{c\in R, k\in\{0,1\}^\lambda}$ with weak pseudorandom outputs. Let further $\rho_0 \in \{0,1\}^\lambda, \rho_1 \in R$.
$\mathsf{PCF.Gen_p}(1^\lambda, \rho_0, \rho_1)$:

1. Set the weak PRF key $k \leftarrow \rho_0$ and $b \leftarrow \rho_1$.
2. Sample a pair of FSS keys $(K_0^{\mathsf{FFS}}, K_1^{\mathsf{FFS}}) \leftarrow \mathsf{FFS.Gen}(1^\lambda, b \cdot F_k)$.
3. Output the keys $\mathsf{k}_0 = (K_0^{\mathsf{FFS}}, k)$ and $\mathsf{k}_1 = (K_1^{\mathsf{FFS}}, b)$.

$\mathsf{PCF.Eval}(\sigma, \mathsf{k}_\sigma, x)$: On input a random x:

- If $\sigma = 0$:
  1. Let $c_0 = -\mathsf{FFS.Eval}(0, K_0^{\mathsf{FFS}}, x)$.
  2. Let $a = F_k(x)$.
  3. Output (a, c_0).
- If $\sigma = 1$:
  1. Let $c_1 = \mathsf{FFS.Eval}(1, K_1^{\mathsf{FFS}}, x)$.
  2. Output (b, c_1).

Theorem 5. *Let $R = R(\lambda)$ be a finite commutative ring. Suppose there exists an FSS scheme for scalar multiples of a family of weak pseudorandom functions $\mathcal{F} = \{F_k : \{0,1\}^\eta \to R\}_{k\in\{0,1\}^\lambda}$. Then, there is a reusable PCF for the VOLE correlation over R as defined in Appendix E.2, given by Construction 5.*

Proof. Boyle et al. showed in their proof of [18, Theorem 5.3] that Construction 5 satisfies pseudorandom $\mathcal{Y}_{\mathsf{VOLE}}$-correlated outputs and security. Although we slightly adapted our definition to consider reusable inputs, their argument still holds. Further, it is easy to see that programmability holds for functions $f_0(\rho_0, x) = F_{\rho_0}(x)$ and $f_1(\rho_1, x) = \rho_1$. Finally, key indistinguishability follows from the secrecy property of the FSS scheme. The secrecy property states that for every function f of the function family, there exists a simulator $\mathcal{S}(1^\lambda)$ such that the output of $\mathcal{S}$ is indistinguishable from the FSS keys generated correctly using the $\mathsf{FFS.Gen}$-algorithm.

To briefly sketch the proof of key indistinguishability, we define a hybrid experiment, where inside the PCF key generation, we use $\mathcal{S}$ to simulate FSS keys. These simulated FSS keys are used inside the PCF key, which is given to $\mathcal{A}_1$. We can show via a reduction to the FSS secrecy that the original $\mathsf{Exp}^{\text{key-ind}}$ game is indistinguishable from the hybrid experiment. For the hybrid experiment, it is easy to see that the adversary can only guess bit b' since the simulated PCF key is independent of $\rho_{1-\sigma}^{(0)}, \rho_{1-\sigma}^{(1)}$ and hence also independent of b. It follows that $\Pr[\mathsf{Exp}^{\text{key-ind}}_{\mathcal{A},\sigma}(\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$.

G PCF-based Threshold Preprocessing Protocol

In this section, we state the PCF-based instantiation of $\mathcal{F}_{\mathsf{Prep}}$. As it is conceptually very similar to the PCG-based instantiation in Section 5, we omit a detailed description and intuition here. We refer the reader to Section 5 for an intuition and a detailed description.

Our protocol $\pi^{\mathsf{PCF}}_{\mathsf{Prep}}$ builds on reusable PCFs for VOLE and OLE correlations. As ssid, which is used to evaluate the PCFs, is provided by the environment, we require strong reusable PCFs.

G.1 Setup Functionality

The setup functionality is identical to $\mathcal{F}^{\mathsf{PCG}}_{\mathsf{Setup}}$ just that the functionality generates PCF keys instead of PCG keys. For the sake of completeness, we formally state $\mathcal{F}^{\mathsf{PCF}}_{\mathsf{Setup}}$ next.

Functionality $\mathcal{F}^{\mathsf{PCF}}_{\mathsf{Setup}}$

Let $(\mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Gen}, \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval})$ be an srPCF for VOLE correlations and let $(\mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Gen}, \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval})$ be an srPCF for OLE correlations. The setup functionality interacts with parties $P_1, \ldots, P_n$ and ideal-world adversary $\mathcal{S}$.

Setup:

Upon receiving $(\mathtt{setup}, \mathsf{sid}, \rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}, \mathsf{sk}_i, \{\mathsf{pk}_\ell^{(i)}\}_{\ell \in [n]})$ from every party P_i send $(\mathtt{setup})$ to $\mathcal{S}$ and do:

1. Check if $g_2^{\mathsf{sk}_\ell} = \mathsf{pk}_\ell^{(i)}$ for every $\ell, i \in [n]$. If the check fails, send $\mathtt{abort}$ to all parties and $\mathcal{S}$. Else, compute for every pair of parties (P_i, P_j):
   (a) $(\mathsf{k}^{\mathsf{VOLE}}_{i,j,0}, \mathsf{k}^{\mathsf{VOLE}}_{i,j,1}) \leftarrow \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Gen}(1^\lambda, \rho_a^{(i)}, \mathsf{sk}_j)$,
   (b) $(\mathsf{k}^{(\mathsf{OLE},1)}_{i,j,0}, \mathsf{k}^{(\mathsf{OLE},1)}_{i,j,1}) \leftarrow \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Gen}(1^\lambda, \rho_a^{(i)}, \rho_s^{(j)})$, and
   (c) $(\mathsf{k}^{(\mathsf{OLE},2)}_{i,j,0}, \mathsf{k}^{(\mathsf{OLE},2)}_{i,j,1}) \leftarrow \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Gen}(1^\lambda, \rho_a^{(i)}, \rho_e^{(j)})$.
2. Send keys $(\mathsf{sid}, \mathsf{k}^{\mathsf{VOLE}}_{i,j,0}, \mathsf{k}^{\mathsf{VOLE}}_{j,i,1}, \mathsf{k}^{(\mathsf{OLE},1)}_{i,j,0}, \mathsf{k}^{(\mathsf{OLE},1)}_{j,i,1}, \mathsf{k}^{(\mathsf{OLE},2)}_{i,j,0}, \mathsf{k}^{(\mathsf{OLE},2)}_{j,i,1})_{j \neq i}$ to every party P_i.

G.2 PCF-based Preprocessing Protocol

In this section, we formally present our PCF-based preprocessing protocol in the $(\mathcal{F}_{\mathsf{KG}}, \mathcal{F}_{\mathsf{Setup}})$-hybrid model.

Construction 6: $\pi^{\mathsf{PCF}}_{\mathsf{Prep}}$

Let $(\mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Gen}, \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval})$ be an srPCF for VOLE correlations and let $(\mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Gen}, \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval})$ be an srPCF for OLE correlations.
We describe the protocol from the perspective of P_i.
Initialization. Upon receiving input $(\mathtt{init}, \mathsf{sid})$, do:

1. Send $(\mathtt{keygen}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{KG}}$.

2. Upon receiving $(\mathsf{sid}, \mathsf{sk}_i, \mathsf{pk}, \{\mathsf{pk}_\ell^{(i)}\}_{\ell\in[n]})$ from $\mathcal{F}_{\mathsf{KG}}$, sample $\rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)} \in \{0,1\}^\lambda$ and send $(\mathtt{setup}, \mathsf{sid}, \rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}, \mathsf{sk}_i, \{\mathsf{pk}_\ell^{(i)}\}_{\ell\in[n]})$ to $\mathcal{F}_{\mathsf{Setup}}$.
3. Upon receiving $(\mathsf{sid}, \mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{k}_{j,i,1}^{\mathsf{VOLE}}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},2)})_{j\neq i}$ from $\mathcal{F}_{\mathsf{Setup}}$, output pk.

Tuple. Upon receiving input $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$, compute:

4. for $j \in \mathcal{T} \setminus \{i\}$:
   (a) $(a_i, c_{i,j,0}^{\mathsf{VOLE}}) = \mathsf{PCF}_{\mathsf{VOLE}}.\mathsf{Eval}(0, \mathsf{k}_{i,j,0}^{\mathsf{VOLE}}, \mathsf{ssid})$,
   (b) $(\mathsf{sk}_i, \mathbf{c}_{j,i,1}^{\mathsf{VOLE}}) = \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(1, \mathsf{k}_{j,i,1}^{\mathsf{VOLE}})$,
   (c) $(a_i, c_{i,j,0}^{(\mathsf{OLE},1)}) = \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval}(0, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},1)}, \mathsf{ssid})$,
   (d) $(s_i, c_{j,i,1}^{(\mathsf{OLE},1)}) = \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval}(1, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},1)}, \mathsf{ssid})$,
   (e) $(a_i, c_{i,j,0}^{(\mathsf{OLE},2)}) = \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval}(0, \mathsf{k}_{i,j,0}^{(\mathsf{OLE},2)}, \mathsf{ssid})$, and
   (f) $(e_i, c_{j,i,1}^{(\mathsf{OLE},2)}) = \mathsf{PCF}_{\mathsf{OLE}}.\mathsf{Eval}(1, \mathsf{k}_{j,i,1}^{(\mathsf{OLE},2)}, \mathsf{ssid})$.
5. $\delta_i = a_i(e_i + L_{i,\mathcal{T}}\mathsf{sk}_i) + \sum_{j\in\mathcal{T}\setminus\{i\}} \left(L_{i,\mathcal{T}} c_{j,i,1}^{\mathsf{VOLE}} - L_{j,\mathcal{T}} c_{i,j,0}^{\mathsf{VOLE}} + c_{j,i,1}^{(\mathsf{OLE},2)} - c_{i,j,0}^{(\mathsf{OLE},2)} \right)$
6. $\alpha_i = a_i s_i + \sum_{j\in\mathcal{T}\setminus\{i\}} \left(c_{j,i,1}^{(\mathsf{OLE},1)} - c_{i,j,0}^{(\mathsf{OLE},1)} \right)$

Finally, output $(\mathsf{sid}, \mathsf{ssid}, a_i, e_i, s_i, \delta_i, \alpha_i)$.

Theorem 6. *Let* $\mathsf{PCF}_{\mathsf{VOLE}}$ *be an srPCF for VOLE correlations and let* $\mathsf{PCF}_{\mathsf{OLE}}$ *be an srPCF for OLE correlations as defined in Appendix E.2. Then, protocol* $\pi_{\mathsf{Prep}}^{\mathsf{PCF}}$ *UC-realizes* $\mathcal{F}_{\mathsf{Prep}}$ *in the* $(\mathcal{F}_{\mathsf{KG}}, \mathcal{F}_{\mathsf{Setup}})$*-hybrid model in the presence of malicious adversaries controlling up to* $t-1$ *parties.*

The proof works analogously to the proof of Theorem 2, which is presented in Appendix L. Therefore, we omit the proof of Theorem 6 for the sake of conciseness.

H Ideal Threshold Signature Functionality

In this section, we state our ideal threshold functionality $\mathcal{F}_{\mathsf{tsig}}$ on which we base our security analysis of the online protocol (cf. Theorem 1). The functionality is presented in the universal composability (UC) framework and we refer the reader to Appendix B for a brief introduction into the UC framework and its notation. $\mathcal{F}_{\mathsf{tsig}}$ is a modification of the functionality proposed by Canetti et al. [29]. First, we allow the parties to specify a set of signers $\mathcal{T}$ during the signing request. This allows us to account for a flexible threshold of signers instead of requiring all n parties to sign. Second, we model the signed message as an array of messages. This change accounts for signature schemes allowing signing k messages simultaneously, such as BBS+. Third, we remove the identifiability property, the key-refresh, and the corruption/decorruption interface. The key-refresh and the corruption/decorruption interface are not required in our scenario as we consider a static adversary in contrast to the mobile adversary in [29]. Fourth, we allow every party to sign only one message per ssid. Finally, at the end of the signing phase, honest parties might output $\mathtt{abort}$ instead of a valid signature. This modification is due to our protocol not providing robustness or identifiable abort.

Next, we state the full formal description of our threshold signature functionality $\mathcal{F}_{\mathsf{tsig}}$.

Functionality $\mathcal{F}_{\mathsf{tsig}}$

The functionality is parameterized by a threshold parameter t. We denote a set of t parties by $\mathcal{T}$. For a specific session id sid, the sub-procedures *Signing* and *Verification* can only be executed once a tuple $(\mathsf{sid}, \mathcal{V})$ is recorded.

Key-generation.

1. Upon receiving $(\texttt{keygen}, \mathsf{sid})$ from some party P_i, interpret $\mathsf{sid} = (\ldots, \mathbf{P})$, where $\mathbf{P} = (P_1, \ldots, P_n)$.
   – If $P_i \in \mathbf{P}$, send to $\mathcal{S}$ and record $(\texttt{keygen}, \mathsf{sid}, i)$.
   – Otherwise ignore the message.
2. Once $(\texttt{keygen}, \mathsf{sid}, i)$ is recorded for all $P_i \in \mathbf{P}$, send $(\texttt{pubkey}, \mathsf{sid})$ to the adversary $\mathcal{S}$ and do:
   (a) Upon receiving $(\texttt{pubkey}, \mathsf{sid}, \mathcal{V})$ from $\mathcal{S}$, record $(\mathsf{sid}, \mathcal{V})$.
   (b) Upon receiving $(\texttt{pubkey}, \mathsf{sid})$ from $P_i \in \mathbf{P}$, output $(\texttt{pubkey}, \mathsf{sid}, \mathcal{V})$ if it is recorded. Else ignore the message.

Signing.

1. Upon receiving $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m} = (m_1, \ldots, m_k))$ with $\mathcal{T} \subseteq \mathbf{P}$, from $P_i \in \mathcal{T}$ and no tuple $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \cdot, \cdot, i)$ is stored, send to $\mathcal{S}$ and record $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, i)$.
2. Upon receiving $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m} = (m_1, \ldots, m_k), i)$ from $\mathcal{S}$, record $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, i)$ if $P_i \in \mathcal{C}$. Else ignore the message.
3. Once $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, i)$ is recorded for all $P_i \in \mathcal{T}$, send $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m})$ to the adversary $\mathcal{S}$.
4. Upon receiving $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \sigma, \mathcal{I})$ from $\mathcal{S}$, where $\mathcal{I} \subseteq \mathcal{T} \setminus \mathcal{C}$, do:
   – If there exists a record $(\mathsf{sid}, \mathbf{m}, \sigma, 0)$, output an error.
   – Else, record $(\mathsf{sid}, \mathbf{m}, \sigma, \mathcal{V}(\mathbf{m}, \sigma))$, send $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \sigma)$ to all $P_i \in \mathcal{T} \setminus (\mathcal{C} \cup \mathcal{I})$ and send $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \texttt{abort})$ to all $P_i \in \mathcal{T} \cap \mathcal{I}$.

Verification.

Upon receiving $(\texttt{verify}, \mathsf{sid}, \mathbf{m} = (m_1, \ldots, m_k), \sigma, \mathcal{V}')$ from a party Q, send the tuple $(\texttt{verify}, \mathsf{sid}, \mathbf{m}, \sigma, \mathcal{V}')$ to $\mathcal{S}$ and do:
– If $\mathcal{V}' = \mathcal{V}$ and a tuple $(\mathsf{sid}, \mathbf{m}, \sigma, \beta')$ is recorded, then set $\beta = \beta'$.
– Else, if $\mathcal{V}' = \mathcal{V}$ and less than t parties in $\mathbf{P}$ are corrupted, set $\beta = 0$ and record $(\mathsf{sid}, \mathbf{m}, \sigma, 0)$.
– Else, set $\beta = \mathcal{V}'(\mathbf{m}, \sigma)$.
Output $(\texttt{verified}, \mathsf{sid}, \mathbf{m}, \sigma, \beta)$ to Q.

I Proof of Theorem 1

This section presents the proof of our online protocol, i.e., Theorem 1.

Proof. We construct a simulator $\mathcal{S}$ that interacts with the environment and the ideal functionality $\mathcal{F}_{\mathsf{tsig}}$. Since the security statement for UC requires that for every real-world adversary $\mathcal{A}$, there is a simulator $\mathcal{S}$, we allow $\mathcal{S}$ to execute $\mathcal{A}$ internally. In the internal execution of $\mathcal{A}$, $\mathcal{S}$ acts as the environment and the honest parties. In particular, $\mathcal{S}$ forwards all messages between its environment and $\mathcal{A}$. The adversary $\mathcal{A}$ creates messages for the corrupted parties. These messages are sent to $\mathcal{S}$ in the internal execution. Note that this scenario also covers dummy adversaries, which just forward messages received from the environment. An output of $\mathcal{S}$ indistinguishable from the output of $\mathcal{A}$ in the real-world execution is created by simulating a protocol transcript towards $\mathcal{A}$ that is indistinguishable from the real-world execution and outputting whatever $\mathcal{A}$ outputs in the simulated execution. Since the protocol $\pi_{\mathsf{TBBS+}}$ is executed in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model, $\mathcal{S}$ impersonates the hybrid functionality $\mathcal{F}_{\mathsf{Prep}}$ in the internal execution.

We start with presenting our simulator $\mathcal{S}$.

<u>Simulator $\mathcal{S}$</u>

KeyGen.

1. Upon receiving $(\texttt{init}, \mathsf{sid})$ from corrupted party P_j, send $(\texttt{keygen}, \mathsf{sid})$ on behalf of P_j to $\mathcal{F}_{\mathsf{tsig}}$.
2. Upon receiving $(\texttt{pubkey}, \mathsf{sid})$ from $\mathcal{F}_{\mathsf{tsig}}$ simulate the initialization phase of $\mathcal{F}_{\mathsf{Prep}}$ to get pk. In particular, sample $\mathsf{sk} \overset{\$}{\leftarrow} \mathbb{Z}_p$ and send $\mathsf{pk} = g_2^{\mathsf{sk}}$ to $\mathcal{A}$.
3. Upon receiving $(\texttt{ok}, \mathsf{Tuple}(\cdot,\cdot,\cdot))$ from $\mathcal{A}$, send $(\texttt{pubkey}, \mathsf{sid}, \mathsf{Verify}_{\mathsf{pk}}(\cdot,\cdot))$ to $\mathcal{F}_{\mathsf{tsig}}$.

Sign.

1. Upon receiving $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m} = \{m_\ell\}_{\ell \in [k]}, i)$ from $\mathcal{F}_{\mathsf{tsig}}$ for honest party P_i, simulate the tuple phase of $\mathcal{F}_{\mathsf{Prep}}$ to get $(a_i, e_i, s_i, \delta_i, \alpha_i)$ for P_i. Then, compute $(A_i := (g_1 \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{a_i} \cdot h_0^{\alpha_i}, \delta_i, e_i, s_i)$ and send it to the corrupted parties in $\mathcal{T}$ in the internal execution.
2. Upon receiving $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m})$ from $\mathcal{Z}$ to corrupted party P_j, send message to P_j in the internal execution an do:
   (a) Upon receiving $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ on behalf of $\mathcal{F}_{\mathsf{Prep}}$ from corrupted party P_j with $j \in \mathcal{T}$ return $(a_j, e_j, s_j, \delta_j, \alpha_j) \leftarrow \mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ to P_j.
   (b) Forward $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, j)$ to $\mathcal{F}_{\mathsf{tsig}}$ and define an empty set $\widehat{\mathcal{I}}_j = \emptyset$ of honest parties that received signature shares from corrupted party P_j.
   (c) Upon receiving $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, A'_{j,i}, \delta'_{j,i}, e'_{j,i}, s'_{j,i})$ from P_j to honest party P_i in the internal execution, add P_i to $\widehat{\mathcal{I}}_j$.
3. Upon receiving $(\texttt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m})$ from $\mathcal{F}_{\mathsf{tsig}}$, do:
   – Use tuple $(a_j, e_j, s_j, \delta_j, \alpha_j)$ to compute honestly generated $(A_j, \delta_j, e_j, s_j)$ for $P_j \in \mathcal{T} \cap \mathcal{C}$. Compute honestly generated signature $\sigma = (A, e, s)$ as honest parties do using $(A_\ell, \delta_\ell, e_\ell, s_\ell)$ for $P_\ell \in \mathcal{T}$.
   – For each honest party P_i recompute signature σ_i obtained by P_i as honest parties do by using $A'_{j,i}, \delta'_{j,i}, e'_{j,i}, s'_{j,i}$ for $P_j \in \mathcal{T} \cap \mathcal{C}$.
   – We define set $\mathcal{I}$ of honest parties that obtained no or an invalid signature. First set, $\mathcal{I} = (\mathcal{T} \setminus \mathcal{C}) \setminus (\bigcap_{j \in \mathcal{T} \cap \mathcal{C}} \widehat{\mathcal{I}}_j)$, i.e., add all honest parties to $\mathcal{I}$ that did not receive signature shares from all corrupted parties in $\mathcal{T}$. Next, compute $\mathcal{I} = \mathcal{I} \cup \{i : \sigma_i \neq \sigma\}$, i.e., add all honest parties that obtained a signature different to the honestly generated signature. If there exists $\sigma_i \neq \sigma$ such that $\mathsf{Verify}_{\mathsf{pk}}(\mathbf{m}, \sigma_i) = 1$ and $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \cdot, \mathbf{m}, \sigma_i, \cdot)$ was not sent to $\mathcal{F}_{\mathsf{tsig}}$ before, output $\texttt{fail}$ and stop the execution.
   – Finally, send $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, \sigma, \mathcal{I})$ to $\mathcal{F}_{\mathsf{tsig}}$.

Verify. Upon receiving $(\texttt{verify}, \mathsf{sid}, \mathbf{m}, \sigma, \mathsf{Verify}_{\mathsf{pk}'}(\cdot,\cdot))$ from $\mathcal{F}_{\mathsf{tsig}}$ check if

– $\mathsf{Verify}_{\mathsf{pk}'}(\cdot,\cdot) = \mathsf{Verify}_{\mathsf{pk}}(\cdot,\cdot)$,
– $(\texttt{sig}, \mathsf{sid}, \mathsf{ssid}, \cdot, \mathbf{m}, \sigma, \cdot)$ was not sent to $\mathcal{F}_{\mathsf{tsig}}$ before
– $\mathsf{Verify}_{\mathsf{pk}}(\mathbf{m}, \sigma) = 1$.

If the checks hold, output $\texttt{fail}$ and stop the execution.

Lemma 1. *If simulator $\mathcal{S}$ does not outputs* `fail`*, protocol $\pi_{\mathsf{TBBS+}}$ UC-realizes $\mathcal{F}_{\mathsf{tsig}}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model in the presence of malicious adversaries controlling up to $t-1$ parties.*

Proof. If the simulator $\mathcal{S}$ does not outputs `fail`, it behaves precisely as the honest parties in real-world execution. Therefore, the simulation is perfect, and no environment can distinguish between the real and ideal worlds.

Lemma 2. *Assuming the strong unforgeability of BBS+, the probability that $\mathcal{S}$ outputs* `fail` *is negligible.*

Proof. We show Lemma 2 via contradiction. Given a real-world adversary $\mathcal{A}$ such that simulator $\mathcal{S}$ outputs `fail` with non-negligible probability, we construct an attacker $\mathcal{B}$ against the strong unforgeability (SUF) of BBS+ with non-negligible success probability. $\mathcal{B}$ simulates the protocol execution towards $\mathcal{A}$ like $\mathcal{S}$ except the following aspects:

1. During the simulation of the initialization phase of $\mathcal{F}_{\mathsf{Prep}}$, instead of sampling $\mathsf{sk} \overset{\$}{\leftarrow} \mathbb{Z}_p$ and computing $\mathsf{pk} = g_2^{\mathsf{sk}}$, $\mathcal{B}$ returns pk^* obtained from the SUF-challenger. Since the SUF-challenger samples the key exactly as the simulator $\mathcal{S}$, this step of the simulations is indistinguishable towards $\mathcal{A}$.
2. During the *Sign* phase, upon receiving $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m}, i)$ from $\mathcal{F}_{\mathsf{tsig}}$ for honest party P_i, the computation of signature shares of the honest parties is modified as follows:
   – Request the signing oracle of the SUF-game on message $\mathbf{m}$ to obtain signature $\sigma = (A, e, s)$. This signature is forwarded to $\mathcal{F}_{\mathsf{tsig}}$ on receiving $(\mathtt{sign}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \mathbf{m})$ from $\mathcal{F}_{\mathsf{tsig}}$.
   – Compute $(a_j, e_j, s_j, \delta_j, \alpha_j) \leftarrow \mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ and (A_j, e_j, s_j) according to the protocol specification for every corrupted party $P_j \in \mathcal{T} \cap \mathcal{C}$.
   – Sample random index $k \overset{\$}{\leftarrow} \mathcal{T} \setminus \mathcal{C}$.
   – For all honest parties except P_k sample random signature share, i.e., $\forall P_i \in (\mathcal{T} \setminus \mathcal{C}) \setminus \{P_k\}$: $(A_i, \delta_i, e_i, s_i) \overset{\$}{\leftarrow} (\mathbb{G}_1, \mathbb{Z}_p, \mathbb{Z}_p, \mathbb{Z}_p)$.
   – For P_k sample random $\delta_k \overset{\$}{\leftarrow} \mathbb{Z}_p$ and compute $e_k = e - \sum_{\ell \in \mathcal{T} \setminus \{k\}} e_\ell$, $s_k = s - \sum_{\ell \in \mathcal{T} \setminus \{k\}} s_\ell$, and
   $$A_k = \frac{A^{\sum_{\ell \in \mathcal{T}} \delta_\ell}}{\prod_{\ell \in \mathcal{T} \setminus \{k\}} A_\ell}.$$

   It is easy to see that e_i and s_i are sampled at random by both, $\mathcal{S}$ and $\mathcal{B}$. Moreover, δ_i is a share of $a(\mathsf{sk} + e)$ in the simulation by $\mathcal{S}$ and since the random value a works as a random mask, it has the same distribution as in the simulation by $\mathcal{B}$. Finally, the A_i values yield a valid signature in $\mathcal{B}$. Therefore, the simulation of the *Sign* phase of $\mathcal{B}$ and $\mathcal{S}$ are indistinguishable to $\mathcal{A}$.

Finally, $\mathcal{B}$ needs to provide a strong forgery to the SUF-challenger. Here, we use the fact that $\mathcal{S}$ outputs `fail` with non-negligible probability either in the *Sign* or the *Verify* phase. As the interaction of $\mathcal{B}$ with $\mathcal{A}$ is indistinguishable, $\mathcal{B}$ outputs `fail` with non-negligible probability as well. Whenever $\mathcal{B}$ outputs `fail`, it forwards the pair $(\mathbf{m}^*, \sigma^*)$ obtained in the *Sign* or *Verify* phase to the SUF-challenger.

It remains to show that $\mathcal{B}$ successfully wins the SUF-game. In order to be a valid forgery, it must hold that (1) $\mathsf{Verify}_{\mathsf{pk}^*}(\mathbf{m}^*, \sigma^*) = 1$ and (2) $(\mathbf{m}^*, \sigma^*)$ was not returned by the signing oracle before. (1) is trivially true, since $\mathcal{B}$ only outputs `fail` if this condition holds. For (2), we note that $\mathcal{A}$ has never seen σ^* as output from $\mathcal{F}_{\mathsf{tsig}}$, since $\mathcal{B}$ checks that $(\mathtt{sig}, \mathsf{sid}, \mathsf{ssid}, \cdot, \mathbf{m}^*, \sigma^*, \cdot)$ was not sent to $\mathcal{F}_{\mathsf{tsig}}$ before. However, it might happen that $\mathcal{B}$ obtained σ^* as response to a signing request for message $\mathbf{m}^*$ without forwarding it the to $\mathcal{F}_{\mathsf{tsig}}$ (this happens if the environment does not instruct all parties in $\mathcal{T}$ to sign). Since the signing oracle samples e and s at random from $\mathbb{Z}_p$, the probability that σ^* was returned by the signing oracle is $\leq \frac{q}{p}$, where q is the number of oracle requests and p is the size of the field. While q is a polynomial, p is exponential in the security parameter. Thus, the probability that σ^* hits an unseen response from the signing oracle is negligible in the security parameter. It follows that $(\mathbf{m}^*, \sigma^*)$ is a valid forgery and $\mathcal{B}$ wins the SUF-game.

Since this contradicts the strong unforgeability of BBS+, it follows that the probability that $\mathcal{S}$ outputs `fail` is negligible.

Combining Lemma 1 and Lemma 2 concludes the proof of Theorem 1.

J Simulator for PCG-based Preprocessing

Here, we state our simulator for proving security of our PCG-based preprocessing. Formally, the security is stated in Theorem 2. We provide a proof sketch of our indistinguishability argument in Appendix K and state the full proof in Appendix L.

Simulator for Preprocessing $\mathcal{S}$

Without loss of generality, we assume the adversary corrupts parties $P_1, \ldots, P_{t-1}$ and parties $P_t, \ldots, P_n$ are honest. $\mathcal{S}$ internally uses adversary $\mathcal{A}$.

Initialization.

1: • Upon receiving $(\texttt{keygen}, \mathsf{sid})$ on behalf of $\mathcal{F}_{\mathsf{KG}}$ from corrupted party P_j, send $(\texttt{init}, \mathsf{sid})$ on behalf of corrupted P_j to $\mathcal{F}_{\mathsf{Prep}}$. Then, wait to receive $(\texttt{corruptedShares}, \mathsf{sid}, \{\mathsf{sk}_j\}_{j \in \mathcal{C}})$ from $\mathcal{A}$.

2: • Upon receiving pk from $\mathcal{F}_{\mathsf{Prep}}$, set $\mathsf{pk}_j = g_2^{\mathsf{sk}_j}$ for $j \in \mathcal{C}$ and compute $\mathsf{pk}_i = \left(\mathsf{pk}/(\mathsf{pk}_1^{L_{1,\mathcal{T}}} \cdot \ldots \cdot \mathsf{pk}_{t-1}^{L_{1,\mathcal{T}}})\right)^{1/L_{i,\mathcal{T}}}$, where $\mathcal{T} := \mathcal{C} \cup \{i\}$, for every honest party P_i. Then, send $(\mathsf{sid}, \mathsf{sk}_j, \mathsf{pk}, \{\mathsf{pk}_\ell\}_{\ell \in [n]})$ to every corrupted party P_j.

• Upon receiving $(\texttt{setup}, \mathsf{sid}, \rho_a^{(j)}, \rho_s^{(j)}, \rho_e^{(j)}, \mathsf{sk}'_j, \{\mathsf{pk}_\ell^{(j)}\}_{\ell \in [n]})$ on behalf of $\mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}}$ from every corrupted party P_j, check that $\mathsf{pk}_\ell^{(j)} = \mathsf{pk}_\ell$ and $g_2^{\mathsf{sk}'_j} = \mathsf{pk}_j$ for $j \in \mathcal{C}$ and $\ell \in [n]$. If any check fails, send $(\texttt{abort}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{Prep}}$. Otherwise sample $\rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}$ and a dummy secret key share $\widehat{\mathsf{sk}}_i$ for every honest party P_i and simulate the computation of $\mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}}$ (i.e., compute all the PCG keys using the values received from the corrupted parties and the values sampled for the honest parties).

3: • Send keys $(\mathsf{sid}, \mathsf{k}_{j,\ell,0}^{\mathsf{VOLE}}, \mathsf{k}_{\ell,j,1}^{\mathsf{VOLE}}, \mathsf{k}_{j,\ell,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{\ell,j,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,\ell,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{\ell,j,1}^{(\mathsf{OLE},2)})_{\ell \neq j}$ to every corrupted party P_j.

• Send $(\mathsf{ok}, \mathsf{Tuple}(\cdot,\cdot,\cdot))$ to $\mathcal{F}_{\mathsf{Prep}}$, where $\mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ computes $(a_j, e_j, s_j, \delta_j, \alpha_j)$ for corrupted party P_j exactly as P_j computes its tuple in the protocol description.

First, expand for every $\ell \in \mathcal{T} \setminus \{j\}$:

$$\begin{aligned}
(\mathbf{a}_j, \mathbf{c}^{\mathsf{VOLE}}_{j,\ell,0}) &= \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(0, \mathsf{k}^{\mathsf{VOLE}}_{j,\ell,0})\,, \\
(\mathsf{sk}_j, \mathbf{c}^{\mathsf{VOLE}}_{\ell,j,1}) &= \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(1, \mathsf{k}^{\mathsf{VOLE}}_{\ell,j,0})\,, \\
(\mathbf{a}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{j,\ell,0}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}^{(\mathsf{OLE},1)}_{j,\ell,0})\,, \\
(\mathbf{s}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{\ell,j,1}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}^{(\mathsf{OLE},1)}_{\ell,j,1})\,, \\
(\mathbf{a}_j, \mathbf{c}^{(\mathsf{OLE},2)}_{j,\ell,0}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}^{(\mathsf{OLE},2)}_{j,\ell,0})\,, \\
(\mathbf{e}_j, \mathbf{c}^{(\mathsf{OLE},2)}_{\ell,j,1}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}^{(\mathsf{OLE},2)}_{\ell,j,1})\,.
\end{aligned}$$

Next, set $a_j = \mathbf{a}_j[\mathsf{ssid}], e_j = \mathbf{e}_j[\mathsf{ssid}], s_j = \mathbf{s}_j[\mathsf{ssid}], c^{\mathsf{VOLE}}_{(j,\ell,0)} = \mathbf{c}^{\mathsf{VOLE}}_{(j,\ell,0)}[\mathsf{ssid}]$, $c^{\mathsf{VOLE}}_{(\ell,j,1)} = \mathbf{c}^{\mathsf{VOLE}}_{(\ell,j,1)}[\mathsf{ssid}]$, $c^{(\mathsf{OLE},d)}_{(j,\ell,0)} = \mathbf{c}^{(\mathsf{OLE},d)}_{(j,\ell,0)}[\mathsf{ssid}]$ and $c^{(\mathsf{OLE},d)}_{(\ell,j,1)} = \mathbf{c}^{(\mathsf{OLE},d)}_{(\ell,j,1)}[\mathsf{ssid}]$ for $\ell \in \mathcal{T} \setminus \{j\}$ and $d \in \{1,2\}$ and compute

$$\begin{aligned}
\alpha_j &= a_j s_j + \sum_{\ell \in \mathcal{T}\setminus\{j\}} c^{(\mathsf{OLE},1)}_{\ell,j,1} - c^{(\mathsf{OLE},1)}_{j,\ell,0}\,, \\
\delta_j &= a_j(L_{j,\mathcal{T}}\mathsf{sk}_j + e_j) \\
&\quad + \sum_{\ell \in \mathcal{T}\setminus\{j\}} \left(L_{j,\mathcal{T}} c^{\mathsf{VOLE}}_{\ell,j,1} - L_{\ell,\mathcal{T}} c^{\mathsf{VOLE}}_{j,\ell,0} + c^{(\mathsf{OLE},2)}_{\ell,j,1} - c^{(\mathsf{OLE},2)}_{j,\ell,0}\right)\,.
\end{aligned}$$

Tuple. Upon receiving $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ from $\mathcal{Z}$ on behalf of corrupted party P_j, forward message $(\texttt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ to $\mathcal{A}$ and output whatever $\mathcal{A}$ outputs.

K Indistinguishability Proof Sketch of Theorem 2

We prove indistinguishability between the ideal-world execution and the real-world execution via a sequence of hybrid experiments. We start with Hybrid_0 which is the ideal-world execution and end up in Hybrid_7 being identical to the real-world execution. By showing indistinguishability between each subsequent pair of hybrids, it follows that the ideal and real-world execution are indistinguishable. In particular, we show indistinguishability between the joint distribution of the adversary's view and the outputs of the honest parties in Hybrid_i and Hybrid_{i+1} for $i = 0, \ldots, 6$. In the following we sketch the proof outline and defer the full proof to Appendix L.

$\underline{\mathsf{Hybrid}_1}$: In this hybrid experiment, we inline the description of the simulator $\mathcal{S}$, the ideal functionality $\mathcal{F}_{\mathsf{Prep}}$ and the outputs of the honest parties. Since this is only a syntactical change, the distribution is identical to the one of Hybrid_0.

$\underline{\mathsf{Hybrid}_2}$: Instead of sampling the secret key sk at random from $\mathbb{Z}_p$, we sample a random polynomial $F(x) \in \mathbb{Z}_p[X]$ of degree $t-1$ such that $F(j) = \mathsf{sk}_j$ for every $j \in \mathcal{C}$. The secret key is then defined as $\mathsf{sk} = F(0)$.

Note that the adversary knows only $t-1$ shares of the polynomial which give no information about sk. This is due to the information-theoretically secrecy of Shamir's secret sharing. It follows that Hybrid_1 and Hybrid_2 are perfectly indistinguishable.

$\underline{\mathsf{Hybrid}_3}$: In this hybrid, we change the way honest parties' secret key shares are defined. Instead of sampling random dummy key shares, we derive the key shares from the polynomial introduced in the last hybrid. In more detail, the key share of honest party P_i is computed as $\mathsf{sk}_i = F(i)$.

This change effects the PCG key generation as the dummy key share is replaced by sk_i for honest party P_i.

To show indistinguishability between Hybrid_2 and Hybrid_3, we reduce to the key indistinguishability property of the $\mathsf{PCG}_{\mathsf{VOLE}}$ primitive. More specifically, we introduce a sequence of intermediate hybrids where we only change the secret key of a single honest party in each step.

$\underline{\mathsf{Hybrid}_4}$: In this hybrid, we change the computation of the honest party P_i's public key share pk_i. Instead of interpolating pk_i it is defined as $\mathsf{pk}_i = g_2^{\mathsf{sk}_i}$. As both ways are equivalent, Hybrid_4 is perfectly indistinguishable from Hybrid_3.

$\underline{\mathsf{Hybrid}_5}$: In this hybrid, we make the sampling of the honest parties' outputs of the tuple phase explicit. To this end, we compute the tuple values in two steps. First, we sample values for a_i, e_i and s_i, then we compute α_i and δ_i. For sampling, we distinguish between two cases. (1) For every pair of two honest parties (P_i, P_ℓ) the values are sampled from $\mathcal{Y}_{\mathsf{VOLE}}$ and $\mathcal{Y}_{\mathsf{OLE}}$. (2) For every pair of one honest party P_i and one corrupted party P_j, we use the reverse-sampling algorithm of the correlations to compute the correlation outputs of the honest party. We illustrate the idea for a_i, s_i and α_i in the following.

After simulating the PCG key generation of $\mathcal{F}^{\mathsf{PCG}}_{\mathsf{Setup}}$, the experiment computes once and stores for every $i, \ell \in ([N] \setminus \mathcal{C})$ with $i \neq \ell$:

$$((\mathbf{a}_i, \mathbf{c}^{(\mathsf{OLE},1)}_{i,\ell,0}), \cdot) \in \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(i)}, \rho_s^{(\ell)}), [N]),$$
$$(\cdot, (\mathbf{s}_i, \mathbf{c}^{(\mathsf{OLE},1)}_{\ell,i,1})) \in \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(\ell)}, \rho_s^{(i)}), [N]),$$

and for every $i \in ([N] \setminus \mathcal{C}), j \in ([N] \cap \mathcal{C})$:

$$(\mathbf{a}_i, \mathbf{c}^{(\mathsf{OLE},1)}_{i,j,0}) \leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(i)}, \rho_s^{(j)}), 0, (\mathbf{s}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{i,j,1}), [N])$$
$$(\mathbf{s}_i, \mathbf{c}^{(\mathsf{OLE},1)}_{j,i,1}) \leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(j)}, \rho_s^{(i)}), 1, (\mathbf{a}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{j,i,0}), [N]),$$

where $(\mathbf{a}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{j,i,0}) = \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}^{(\mathsf{OLE},1)}_{j,i,0})$ and $(\mathbf{s}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{i,j,1}) = \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}^{(\mathsf{OLE},1)}_{i,j,1})$.

Then, during the tuple phase, for every $j \in \mathcal{T} \setminus \{i\}$ let $a_i = \mathbf{a}_i[\mathsf{ssid}], s_i = \mathbf{s}_i[\mathsf{ssid}], c^{(\mathsf{OLE},1)}_{i,j,0} = \mathbf{c}^{(\mathsf{OLE},1)}_{i,j,0}[\mathsf{ssid}]$, and $c^{(\mathsf{OLE},1)}_{j,i,1} = \mathbf{c}^{(\mathsf{OLE},1)}_{j,i,1}[\mathsf{ssid}]$ and compute

$$\alpha_i = a_i s_i + \sum_{\ell \in \mathcal{T} \setminus \{i\}} c^{(\mathsf{OLE},1)}_{\ell,i,1} - c^{(\mathsf{OLE},1)}_{i,\ell,0}.$$

Similar process is done for the computation of δ_i and e_i. A straightforward calculation shows that resulting tuple values satisfy correlation (4). Note that the reverse-sampling and the correlation sampling outputs uniform correlation outputs and hence the correlation is identically distributed as in Hybrid_4. It follows that the view of the environment is indistinguishable in Hybrid_4 and Hybrid_5.

$\underline{\mathsf{Hybrid}_6}$: Now, we replace the sampling of correlation outputs for calculating honest parties' tuples (cf. case (1) of previous hybrid) with the expansion of the PCG keys, i.e., instead of using outputs of the $\mathcal{Y}_{\mathsf{VOLE}}$ and $\mathcal{Y}_{\mathsf{OLE}}$ correlations, we run the $\mathsf{PCG}_{\mathsf{VOLE}}$ and $\mathsf{PCG}_{\mathsf{OLE}}$ expansions. For running the PCG expansions, we use the keys obtained during the simulation of $\mathcal{F}^{\mathsf{PCG}}_{\mathsf{Setup}}$ in step (2).

Indistinguishability between Hybrid_5 and Hybrid_6 can be shown via reductions to the pseudorandom $\mathcal{Y}_{\mathsf{VOLE}}$-correlated output property of the $\mathsf{PCG}_{\mathsf{VOLE}}$ primitive and to the pseudorandom $\mathcal{Y}_{\mathsf{OLE}}$-correlated output property of the $\mathsf{PCG}_{\mathsf{OLE}}$ primitive, respectively. More precisely, a series of intermediate hybrids can be introduce, where in each hop only a single correlation output is replaced by the output of PCG expansions.

$\underline{\mathsf{Hybrid}_7}$: Finally, we replace the reverse-sampling in case (2) of Hybrid_5 with the PCG expansion. The indistinguishability between Hybrid_6 and Hybrid_7 can be shown via a reduction to the security property of the rPCGs.

Hybrid_7 is the real-world execution, which concludes the proof.

L Full Indistinguishability Proof of Theorem 2

In this section, we provide the full indistinguishability proof of Theorem 2. The simulator is given in Appendix J.

$\underline{\mathsf{Hybrid}_0}$: The initial experiment Hybrid_0 denotes the ideal-world execution where simulator $\mathcal{S}$ is interacting with the corrupted parties, ideal functionality $\mathcal{F}_{\mathsf{Prep}}$ and internally runs real-world adversary $\mathcal{A}$.

$\underline{\mathsf{Hybrid}_1}$: In this hybrid, we inline the description of the simulator $\mathcal{S}$, the ideal functionality $\mathcal{F}_{\mathsf{Prep}}$ and the outputs of the honest parties. Since this is only a syntactical change, the joint distribution of the adversary's view and the output of the honest parties is identical to the one of Hybrid_0. We state Hybrid_1 as the starting point, and emphasize only on the changes in the following hybrids.

$\underline{\mathsf{Hybrid}_1}$

Without loss of generality, we assume the adversary corrupts parties $P_1, \ldots, P_{t-1}$ and parties $P_t, \ldots, P_n$ are honest. $\mathcal{S}$ internally uses adversary $\mathcal{A}$.

Initialization.

1: • Upon receiving $(\texttt{keygen}, \mathsf{sid})$ on behalf of $\mathcal{F}_{\mathsf{KG}}$ from corrupted party P_j, store $(\texttt{init}, \mathsf{sid}, P_j)$. Then, wait to receive $(\texttt{corruptedShares}, \mathsf{sid}, \{\mathsf{sk}_j\}_{j \in \mathcal{C}})$ from $\mathcal{A}$.

• Upon receiving $(\texttt{init}, \mathsf{sid})$ from every honest party, sample the secret key $\mathsf{sk} \overset{\$}{\leftarrow} \mathbb{Z}_p$ and set $\mathsf{pk} = g_2^{\mathsf{sk}}$. Further, set $\mathsf{pk}_j = g_2^{\mathsf{sk}_j}$ for $j \in \mathcal{C}$ and compute $\mathsf{pk}_i = \left(\mathsf{pk}/(\mathsf{pk}_1^{L_{1,\mathcal{T}}} \cdot \ldots \cdot \mathsf{pk}_{t-1}^{L_{1,\mathcal{T}}})\right)^{1/L_{i,\mathcal{T}}}$, where $\mathcal{T} := \mathcal{C} \cup \{i\}$, for every honest party P_i.

2: • Send $(\mathsf{sid}, \mathsf{sk}_j, \mathsf{pk}, \{\mathsf{pk}_\ell\}_{\ell \in [n]})$ to every corrupted party P_j.

• Upon receiving $(\texttt{setup}, \mathsf{sid}, \rho_a^{(j)}, \rho_s^{(j)}, \rho_e^{(j)}, \mathsf{sk}'_j, \{\mathsf{pk}_\ell^{(j)}\}_{\ell \in [n]})$ on behalf of $\mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}}$ from every corrupted party P_j, check that $\mathsf{pk}_\ell^{(j)} = \mathsf{pk}_\ell$ and $g_2^{\mathsf{sk}'_j} = \mathsf{pk}_j$ for $j \in \mathcal{C}$ and $\ell \in [n]$. If any check fails, honest parties output $\texttt{abort}$. Otherwise sample $\rho_a^{(i)}, \rho_s^{(i)}, \rho_e^{(i)}$ and a dummy secret key share $\widehat{\mathsf{sk}}_i$ for every honest party P_i and simulate the computation of $\mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}}$ (i.e., compute all the PCG keys using the values received from the corrupted parties and the values sampled for the honest parties).

3: • Send keys $(\mathsf{sid}, \mathsf{k}_{j,\ell,0}^{\mathsf{VOLE}}, \mathsf{k}_{\ell,j,1}^{\mathsf{VOLE}}, \mathsf{k}_{j,\ell,0}^{(\mathsf{OLE},1)}, \mathsf{k}_{\ell,j,1}^{(\mathsf{OLE},1)}, \mathsf{k}_{j,\ell,0}^{(\mathsf{OLE},2)}, \mathsf{k}_{\ell,j,1}^{(\mathsf{OLE},2)})_{\ell \neq j}$ to every corrupted party P_j.

• Store $(\texttt{ok}, \mathsf{Tuple}(\cdot,\cdot,\cdot))$, where $\mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ computes $(a_j, e_j, s_j, \delta_j, \alpha_j)$ for corrupted party P_j exactly as P_j computes its tuple in the protocol description.

First, expand for every $\ell \in \mathcal{T} \setminus \{j\}$:

$$
\begin{aligned}
(\mathbf{a}_j, \mathbf{c}^{\mathsf{VOLE}}_{j,\ell,0}) &= \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(0, \mathsf{k}^{\mathsf{VOLE}}_{j,\ell,0}),\\
(\mathsf{sk}_j, \mathbf{c}^{\mathsf{VOLE}}_{\ell,j,1}) &= \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(1, \mathsf{k}^{\mathsf{VOLE}}_{\ell,j,0}),\\
(\mathbf{a}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{j,\ell,0}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}^{(\mathsf{OLE},1)}_{j,\ell,0}),\\
(\mathbf{s}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{\ell,j,1}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}^{(\mathsf{OLE},1)}_{\ell,j,1}),\\
(\mathbf{a}_j, \mathbf{c}^{(\mathsf{OLE},2)}_{j,\ell,0}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}^{(\mathsf{OLE},2)}_{j,\ell,0}),\\
(\mathbf{e}_j, \mathbf{c}^{(\mathsf{OLE},2)}_{\ell,j,1}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}^{(\mathsf{OLE},2)}_{\ell,j,1}).
\end{aligned}
$$

Next, set $a_j = \mathbf{a}_j[\mathsf{ssid}], e_j = \mathbf{e}_j[\mathsf{ssid}], s_j = \mathbf{s}_j[\mathsf{ssid}], c^{\mathsf{VOLE}}_{(j,\ell,0)} = \mathbf{c}^{\mathsf{VOLE}}_{(j,\ell,0)}[\mathsf{ssid}]$, $c^{\mathsf{VOLE}}_{(\ell,j,1)} = \mathbf{c}^{\mathsf{VOLE}}_{(\ell,j,1)}[\mathsf{ssid}]$, $c^{(\mathsf{OLE},d)}_{(j,\ell,0)} = \mathbf{c}^{(\mathsf{OLE},d)}_{(j,\ell,0)}[\mathsf{ssid}]$ and $c^{(\mathsf{OLE},d)}_{(\ell,j,1)} = \mathbf{c}^{(\mathsf{OLE},d)}_{(\ell,j,1)}[\mathsf{ssid}]$ for $\ell \in \mathcal{T} \setminus \{j\}$ and $d \in \{1,2\}$ and compute

$$
\begin{aligned}
\alpha_j &= a_j s_j + \sum_{\ell \in \mathcal{T}\setminus\{j\}} c^{(\mathsf{OLE},1)}_{\ell,j,1} - c^{(\mathsf{OLE},1)}_{j,\ell,0},\\
\delta_j &= a_j(L_{j,\mathcal{T}}\mathsf{sk}_j + e_j)\\
&\quad + \sum_{\ell \in \mathcal{T}\setminus\{j\}} \left(L_{j,\mathcal{T}} c^{\mathsf{VOLE}}_{\ell,j,1} - L_{\ell,\mathcal{T}} c^{\mathsf{VOLE}}_{j,\ell,0} + c^{(\mathsf{OLE},2)}_{\ell,j,1} - c^{(\mathsf{OLE},2)}_{j,\ell,0}\right).
\end{aligned}
$$

- The honest parties $P_t, \ldots, P_n$ output pk.

Tuple.

– Upon receiving $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ from $\mathcal{Z}$ on behalf of corrupted party P_j, forward message $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ to $\mathcal{A}$ and output whatever $\mathcal{A}$ outputs.
– Upon receiving $(\mathtt{tuple}, \mathsf{sid}, \mathsf{ssid}, \mathcal{T})$ from $\mathcal{Z}$ on behalf of honest party P_i, if $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \{(a_\ell, e_\ell, s_\ell, \delta_\ell, \alpha_\ell)\}_{\ell \in \mathcal{T}})$ is stored, output $(\mathsf{sid}, \mathsf{ssid}, a_i, e_i, s_i, \delta_i, \alpha_i)$. Otherwise, compute $(a_j, e_j, s_j, \delta_j, \alpha_j) \leftarrow \mathsf{Tuple}(\mathsf{ssid}, \mathcal{T}, j)$ for every corrupted party P_j where $j \in \mathcal{C} \cap \mathcal{T}$ and sample $a, e, s \overset{\$}{\leftarrow} \mathbb{Z}_p$ and tuples $(a_i, e_i, s_i, \delta_i, \alpha_i)$ over $\mathbb{Z}_p$ for $i \in \mathcal{H} \cap \mathcal{T}$ such that

$$
\sum_{\ell \in \mathcal{T}} a_\ell = a \qquad \sum_{\ell \in \mathcal{T}} e_\ell = e \qquad \sum_{\ell \in \mathcal{T}} s_\ell = s
$$
$$
\sum_{\ell \in \mathcal{T}} \delta_\ell = a(\mathsf{sk} + e) \qquad \sum_{\ell \in \mathcal{T}} \alpha_\ell = as
$$

Store $(\mathsf{sid}, \mathsf{ssid}, \mathcal{T}, \{(a_\ell, e_\ell, s_\ell, \delta_\ell, \alpha_\ell)\}_{\ell \in \mathcal{T}})$ and honest party P_i outputs $(\mathsf{sid}, \mathsf{ssid}, a_i, e_i, s_i, \delta_i, \alpha_i)$.

Hybrid$_2$: In this hybrid, we change the sampling of the secret key sk. Instead of sampling sk in step 1 from $\mathbb{Z}_p$, we sample a random polynomial $F \in \mathbb{Z}_p[X]$ of degree $t-1$ such that $F(j) = \mathsf{sk}_j$ for every $j \in \mathcal{C}$. Further, we define $\mathsf{sk} = F(0)$. Since the polynomial is of degree $t-1$, t evaluation points are required to fully determine $F(x)$. As the adversary knows only $t-1$ shares, it cannot learn anything about sk. In detail, for every $\mathsf{sk}' \in \mathbb{Z}_p$ there exists a t-th share that defined the polynomial $F(x)$ such that $F(x) = \mathsf{sk}'$. It follows that the views of the adversary are distributed identically and hence Hybrid_2 and Hybrid_3 are perfectly indistinguishable.

$\underline{\mathsf{Hybrid}_3}$: Next, we use the polynomial $F(x)$ sampled in step 1 to determine the honest parties' secret key shares. In particular, for every honest party P_i the experiment samples $\mathsf{sk}_i = F(i)$. The secret key shares $\{\mathsf{sk}_i\}_{i \in \mathcal{H}}$ are then used for the simulation of $\mathcal{F}_{\mathsf{Setup}}^{\mathsf{PCG}}$ instead of the dummy key shares. In particular, the correctly sampled key shares of the honest parties are used as input to $\mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Gen}$ whenever a secret key share of the honest party is used. Since the experiment does not use the dummy key shares at all after these changes, we remove them completely. Note that the sampling of the honest parties' key shares and the generation of the PCG keys are exactly as in the real-world execution.

Indistinguishability between Hybrid_2 and Hybrid_3 can be shown via a series of reductions to the key indistinguishability property of the VOLE PCG. We briefly sketch the proof outline in the following. We define intermediate hybrids $\mathsf{Hybrid}_{2,\ell,k}$ for $\ell \in \{0, \ldots, n-(t-1)\}$ and $k \in [n]$, which only differ in the honest parties' key shares that are used in the generation of the VOLE PCG keys. Recall that for every party P_ℓ we generate a VOLE PCG for every other party P_k, where P_ℓ uses its secret key shares as input. We define $\mathsf{Hybrid}_{2,\ell,k}$ such that the key shares derived from polynomial $F(x)$ are used for the first ℓ honest parties in all VOLE PCG instances and for the $(\ell+1)$-th honest party in the VOLE PCG instances with the first k other parties. For all other VOLE PCG instances, the dummy key shares are used for the honest parties' key shares.

Note that $\mathsf{Hybrid}_{2,0,0} = \mathsf{Hybrid}_2$ and $\mathsf{Hybrid}_{2,n-(t-1),n} = \mathsf{Hybrid}_3$. To show indistinguishability between $\mathsf{Hybrid}_{2,\ell,k}$ and $\mathsf{Hybrid}_{2,\ell,k+1}$ for every $\ell \in \{0, \ldots, n-(t-1)\}$, we make a reduction to the key indistinguishability property of the VOLE PCG. In particular, we construct an adversary $\mathcal{A}^{\mathsf{key-ind}}$ from a distinguisher $\mathcal{D}_\ell$ which distinguishes between $\mathsf{Hybrid}_{2,\ell,k}$ and $\mathsf{Hybrid}_{2,\ell,k+1}$. Upon receiving the shares of the corrupted parties in the hybrid execution, $\mathcal{A}^{\mathsf{key-ind}}$ forwards the key share of the $k+1$-th corrupted party to the security game. Then, the security game samples two possible key shares for the ℓ-th honest party $\rho_1^{(0)}, \rho_1^{(1)}$, uses one of them in the VOLE PCG key generation and sends the key k_1 for the corrupted party and $\rho_1^{(0)}$ to $\mathcal{A}^{\mathsf{key-ind}}$. Next, $\mathcal{A}^{\mathsf{key-ind}}$ continues the simulation of hybrid $\mathsf{Hybrid}_{2,\ell,k}$ or $\mathsf{Hybrid}_{2,\ell,k+1}$ by sampling the polynomial $F(x)$ using the corrupted key shares and $\rho_1^{(0)}$. Since $\rho_1^{(0)}$ is a random value in $\mathbb{Z}_p$, $F(x)$ is also a random polynomial. Finally, $\mathcal{A}^{\mathsf{key-ind}}$ uses k_1 as the output of the simulation of $\mathcal{F}_{\mathsf{Setup}}$.

If k_1 was sampled using $\rho_1^{(0)}$, then the simulated experiment is identical to $\mathsf{Hybrid}_{2,\ell,k+1}$ and otherwise it is identical to $\mathsf{Hybrid}_{2,\ell,k}$. It is easy to see that a successful distinguisher between these two hybrids allows to easily win the key indistinguishability game. Since we assume the VOLE PCG to satisfy the key indistinguishability property, this leads to a contradiction. Thus, the two hybrids are indistinguishable.

$\underline{\mathsf{Hybrid}_4}$: In this hybrid, we derive the honest parties public key shares pk_i from the secret key shares sk_i instead of interpolating them from pk and the corrupted shares. More precisely, in Hybrid_3 the public key share of honest party P_i was computed as

$$\mathsf{pk}_i = \left(\mathsf{pk}/(\mathsf{pk}_1^{L_{1,\mathcal{T}}} \cdot \ldots \cdot \mathsf{pk}_{t-1}^{L_{1,\mathcal{T}}})\right)^{1/L_{i,\mathcal{T}}} ,$$

where $\mathcal{T} := \mathcal{C} \cup \{i\}$. In Hybrid_4 the public key share is instead computed as $\mathsf{pk}_i = g_2^{\mathsf{sk}_i}$. We show that both definitions are equivalent.

To this end, note that $\mathsf{sk} = \sum_{\ell \in \mathcal{T}} L_{\ell,\mathcal{T}} \mathsf{sk}_\ell$ for every set $\mathcal{T}$ of size t, $\mathsf{pk} = g_2^{\mathsf{sk}}$ and $\mathsf{pk}_j = g_2^{\mathsf{sk}_j}$ for $j \in \mathcal{C}$. Using this equation we get for $\mathcal{T} = \mathcal{C} \cup \{i\}$

$$\begin{aligned}
\mathsf{pk}_i &= \left(\frac{\mathsf{pk}}{\mathsf{pk}_1^{L_{1,\mathcal{T}}} \cdot \ldots \cdot \mathsf{pk}_{t-1}^{L_{1,\mathcal{T}}}} \right)^{1/L_{i,\mathcal{T}}} \\
\Leftrightarrow \mathsf{pk}_i &= \left(\frac{g_2^{\mathsf{sk}}}{g_2^{L_{1,\mathcal{T}}\mathsf{sk}_1} \cdot \ldots \cdot g_2^{L_{1,\mathcal{T}}\mathsf{sk}_{t-1}}} \right)^{1/L_{i,\mathcal{T}}} \\
\Leftrightarrow \mathsf{pk}_i &= \left(\frac{g_2^{\sum_{\ell \in \mathcal{T}} L_{\ell,\mathcal{T}}\mathsf{sk}_\ell}}{g_2^{L_{1,\mathcal{T}}\mathsf{sk}_1} \cdot \ldots \cdot g_2^{L_{1,\mathcal{T}}\mathsf{sk}_{t-1}}} \right)^{1/L_{i,\mathcal{T}}} \\
\Leftrightarrow \mathsf{pk}_i &= \left(g_2^{L_{i,\mathcal{T}}\mathsf{sk}_i} \right)^{1/L_{i,\mathcal{T}}} \\
\Leftrightarrow \mathsf{pk}_i &= g_2^{\mathsf{sk}_i}
\end{aligned}$$

As public key shares are equivalent in both hybrids, the view of the adversary is identical distributed. Hence, Hybrid_3 and Hybrid_4 are perfectly indistinguishable.

$\underline{\mathsf{Hybrid}_5}$: In this hybrid, we derive the sampling the honest parties' outputs of the tuple phase from correlation samples and reverse sampling. To this end, we distinguish two cases. (1) For every pair of honest parties (P_i, P_ℓ), the values are sampled from $\mathcal{Y}_{\mathsf{VOLE}}$ and $\mathcal{Y}_{\mathsf{OLE}}$. (2) For every pair of one honest party P_i and one corrupted party P_j, we take the output of P_j's PCG expansion and reverse-sample the output of the honest party. More specifically, after simulating the PCG key generation of $\mathcal{F}^{\mathsf{PCG}}_{\mathsf{Setup}}$, the experiment computes once and stores for every $i, \ell \in ([N] \setminus \mathcal{C})$ with $i \neq \ell$:

$$\begin{aligned}
((\mathbf{a}_i, \mathbf{c}^{\mathsf{VOLE}}_{i,\ell,0}), \cdot) &\in \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(i)}, \mathsf{sk}_\ell)), [N])\,, \\
(\cdot, (\mathsf{sk}_i, \mathbf{c}^{\mathsf{VOLE}}_{\ell,i,1})) &\in \mathcal{Y}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(\ell)}, \mathsf{sk}_i), [N])\,, \\
((\mathbf{a}_i, \mathbf{c}^{(\mathsf{OLE},1)}_{i,\ell,0}), \cdot) &\in \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(i)}, \rho_s^{(\ell)}), [N])\,, \\
(\cdot, (\mathbf{s}_i, \mathbf{c}^{(\mathsf{OLE},1)}_{\ell,i,1})) &\in \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(\ell)}, \rho_s^{(i)}), [N])\,, \\
((\mathbf{a}_i, \mathbf{c}^{(\mathsf{OLE},2)}_{i,\ell,0}), \cdot) &\in \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(i)}, \rho_e^{(\ell)}), [N])\,, \\
(\cdot, (\mathbf{e}_i, \mathbf{c}^{(\mathsf{OLE},2)}_{\ell,i,1})) &\in \mathcal{Y}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(\ell)}, \rho_e^{(i)}), [N])\,,
\end{aligned}$$

and for every $i \in ([N] \setminus \mathcal{C}), j \in ([N] \cap \mathcal{C})$:

$$\begin{aligned}
(\mathbf{a}_i, \mathbf{c}^{\mathsf{VOLE}}_{i,j,0}) &\leftarrow \mathsf{RSample}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(i)}, \mathsf{sk}_j), 0, (\mathsf{sk}_j, \mathbf{c}^{\mathsf{VOLE}}_{i,j,1}), [N]) \\
(\mathsf{sk}_i, \mathbf{c}^{\mathsf{VOLE}}_{j,i,1}) &\leftarrow \mathsf{RSample}_{\mathsf{VOLE}}(1^\lambda, (\rho_a^{(j)}, \mathsf{sk}_i), 1, (\mathbf{a}_j, \mathbf{c}^{\mathsf{VOLE}}_{j,i,0}), [N])\,, \\
(\mathbf{a}_i, \mathbf{c}^{(\mathsf{OLE},1)}_{i,j,0}) &\leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(i)}, \rho_s^{(j)}), 0, (\mathbf{s}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{i,j,1}), [N]) \\
(\mathbf{s}_i, \mathbf{c}^{(\mathsf{OLE},1)}_{j,i,1}) &\leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(j)}, \rho_s^{(i)}), 1, (\mathbf{a}_j, \mathbf{c}^{(\mathsf{OLE},1)}_{j,i,0}), [N])\,, \\
(\mathbf{a}_i, \mathbf{c}^{(\mathsf{OLE},2)}_{i,j,0}) &\leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(i)}, \rho_e^{(j)}), 0, (\mathbf{e}_j, \mathbf{c}^{(\mathsf{OLE},2)}_{i,j,1}), [N]) \\
(\mathbf{e}_i, \mathbf{c}^{(\mathsf{OLE},2)}_{j,i,1}) &\leftarrow \mathsf{RSample}_{\mathsf{OLE}}(1^\lambda, (\rho_a^{(j)}, \rho_e^{(i)}), 1, (\mathbf{a}_j, \mathbf{c}^{(\mathsf{OLE},2)}_{j,i,0}), [N])\,,
\end{aligned}$$

where

$$\begin{aligned}(\mathbf{a}_j, \mathbf{c}_{j,i,0}^{\mathsf{VOLE}}) &= \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(0, \mathsf{k}_{j,i,0}^{\mathsf{VOLE}})\,,\\ (\mathsf{sk}_j, \mathbf{c}_{i,j,1}^{\mathsf{VOLE}}) &= \mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(1, \mathsf{k}_{i,j,1}^{\mathsf{VOLE}})\,,\\ (\mathbf{a}_j, \mathbf{c}_{j,i,0}^{(\mathsf{OLE},1)}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}_{j,i,0}^{(\mathsf{OLE},1)})\,,\\ (\mathbf{s}_j, \mathbf{c}_{i,j,1}^{(\mathsf{OLE},1)}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}_{i,j,1}^{(\mathsf{OLE},1)})\,,\\ (\mathbf{a}_j, \mathbf{c}_{j,i,0}^{(\mathsf{OLE},2)}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(0, \mathsf{k}_{j,i,0}^{(\mathsf{OLE},2)})\,,\\ (\mathbf{e}_j, \mathbf{c}_{i,j,1}^{(\mathsf{OLE},2)}) &= \mathsf{PCG}_{\mathsf{OLE}}.\mathsf{Expand}(1, \mathsf{k}_{i,j,1}^{(\mathsf{OLE},2)})\,,\end{aligned}$$

Then, during the tuple phase, for every $j \in \mathcal{T} \setminus \{i\}$ let $a_i = \mathbf{a}_i[\mathsf{ssid}], e_i = \mathbf{e}_i[\mathsf{ssid}], s_i = \mathbf{s}_i[\mathsf{ssid}], c_{i,j,0}^{\mathsf{VOLE}} = \mathbf{c}_{i,j,0}^{\mathsf{VOLE}}[\mathsf{ssid}], c_{j,i,1}^{\mathsf{VOLE}} = \mathbf{c}_{j,i,1}^{\mathsf{VOLE}}[\mathsf{ssid}], c_{i,j,0}^{(\mathsf{OLE},1)} = \mathbf{c}_{i,j,0}^{(\mathsf{OLE},1)}[\mathsf{ssid}], c_{j,i,1}^{(\mathsf{OLE},1)} = \mathbf{c}_{j,i,1}^{(\mathsf{OLE},1)}[\mathsf{ssid}], c_{i,j,0}^{(\mathsf{OLE},2)} = \mathbf{c}_{i,j,0}^{(\mathsf{OLE},2)}[\mathsf{ssid}]$, and $c_{j,i,1}^{(\mathsf{OLE},2)} = \mathbf{c}_{j,i,1}^{(\mathsf{OLE},2)}[\mathsf{ssid}]$ and compute according to the protocol specification

$$\begin{aligned}\alpha_i =& a_i s_i + \sum_{\ell \in \mathcal{T}\setminus\{i\}} c_{\ell,i,1}^{(\mathsf{OLE},1)} - c_{i,\ell,0}^{(\mathsf{OLE},1)}\\ \delta_i =& a_i(e_i + L_{i,\mathcal{T}}\mathsf{sk}_i)\\ &+ \sum_{\ell \in \mathcal{T}\setminus\{i\}} \left(L_{i,\mathcal{T}} c_{\ell,i,1}^{\mathsf{VOLE}} - L_{\ell,\mathcal{T}} c_{i,\ell,0}^{\mathsf{VOLE}} + c_{\ell,i,1}^{(\mathsf{OLE},2)} - c_{i,\ell,0}^{(\mathsf{OLE},2)}\right)\,.\end{aligned}$$

We show that the resulting tuple outputs satisfy the same correlation as before. In particular, we show $\sum_{\ell\in\mathcal{T}} \alpha_\ell = as$ and $\sum_{\ell\in\mathcal{T}} \delta_\ell = a(\mathsf{sk} + e)$, where $a = \sum_{\ell\in\mathcal{T}} a_\ell = \sum_{\ell\in\mathcal{T}} F_{\rho_a^{(\ell)}}(x)$, $e = \sum_{\ell\in\mathcal{T}} e_\ell = \sum_{\ell\in\mathcal{T}} F_{\rho_e^{(\ell)}}(x)$ and $s = \sum_{\ell\in\mathcal{T}} s_\ell = \sum_{\ell\in\mathcal{T}} F_{\rho_s^{(\ell)}}(x)$. First, we show $\sum_{\ell\in\mathcal{T}} \alpha_\ell = as$:

$$\begin{aligned}\sum_{\ell\in\mathcal{T}} \alpha_\ell &= \sum_{\ell\in\mathcal{T}} \left(a_\ell s_\ell + \sum_{k\in\mathcal{T}\setminus\{\ell\}} (c_{k,\ell,1}^{(\mathsf{OLE},1)} - c_{\ell,k,0}^{(\mathsf{OLE},1)})\right)\\ &= \sum_{\ell\in\mathcal{T}} a_\ell s_\ell + \sum_{\ell\in\mathcal{T}} \sum_{k\in\mathcal{T}\setminus\{\ell\}} \left(c_{k,\ell,1}^{(\mathsf{OLE},1)} - c_{k,\ell,0}^{(\mathsf{OLE},1)}\right)\\ &= \sum_{\ell\in\mathcal{T}} a_\ell s_\ell + \sum_{\ell\in\mathcal{T}} \sum_{k\in\mathcal{T}\setminus\{\ell\}} \left(F_{\rho_a^{(k)}}(x) \cdot F_{\rho_s^{(\ell)}}(x)\right)\\ &= \sum_{\ell\in\mathcal{T}} a_\ell s_\ell + \sum_{\ell\in\mathcal{T}} \sum_{k\in\mathcal{T}\setminus\{\ell\}} a_k s_\ell\\ &= \sum_{\ell\in\mathcal{T}} \sum_{k\in\mathcal{T}} a_k s_\ell = \sum_{\ell\in\mathcal{T}} a_k \sum_{k\in\mathcal{T}} s_\ell\\ &= as\end{aligned}$$

Next, we show $\sum_{\ell\in\mathcal{T}}\delta_\ell = a(\mathsf{sk}+e)$:

$$\begin{aligned}
\sum_{\ell\in\mathcal{T}}\delta_\ell &= \sum_{\ell\in\mathcal{T}}\left(a_\ell(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+e_\ell)+\sum_{k\in\mathcal{T}\setminus\{\ell\}}L_{\ell,\mathcal{T}}c^{\mathsf{VOLE}}_{k,\ell,1}-L_{k,\mathcal{T}}c^{\mathsf{VOLE}}_{\ell,k,0}+c^{(\mathsf{OLE},2)}_{k,\ell,1}-c^{(\mathsf{OLE},2)}_{\ell,k,0}\right)\\
&=\sum_{\ell\in\mathcal{T}}a_\ell(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+e_\ell)+\sum_{\ell\in\mathcal{T}}\sum_{k\in\mathcal{T}\setminus\{\ell\}}L_{\ell,\mathcal{T}}c^{\mathsf{VOLE}}_{k,\ell,1}-L_{\ell,\mathcal{T}}c^{\mathsf{VOLE}}_{k,\ell,0}+c^{(\mathsf{OLE},2)}_{k,\ell,1}-c^{(\mathsf{OLE},2)}_{k,\ell,0}\\
&=\sum_{\ell\in\mathcal{T}}a_\ell(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+e_\ell)+\sum_{\ell\in\mathcal{T}}\sum_{k\in\mathcal{T}\setminus\{\ell\}}L_{\ell,\mathcal{T}}a_k\mathsf{sk}_\ell+a_ke_\ell\\
&=\sum_{\ell\in\mathcal{T}}a_\ell(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+e_\ell)+\sum_{\ell\in\mathcal{T}}\sum_{k\in\mathcal{T}\setminus\{\ell\}}a_k(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+e_\ell)\\
&=\sum_{\ell\in\mathcal{T}}\sum_{k\in\mathcal{T}}a_k(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+e_\ell)\\
&=\sum_{k\in\mathcal{T}}\sum_{\ell\in\mathcal{T}}a_k(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+e_\ell)\\
&=\sum_{k\in\mathcal{T}}a_k\sum_{\ell\in\mathcal{T}}(L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+e_\ell)\\
&=\sum_{k\in\mathcal{T}}a_k\left(\sum_{\ell\in\mathcal{T}}L_{\ell,\mathcal{T}}\mathsf{sk}_\ell+\sum_{\ell\in\mathcal{T}}e_\ell\right)\\
&=a(\mathsf{sk}+e)
\end{aligned}$$

As the tuple values of the honest parties still satisfy the same correlation as in Hybrid_4, Hybrid_4 and Hybrid_5 are indistinguishable. Note that the reverse-sampling and the correlation sampling outputs uniform correlation outputs and hence the correlation is identically distributed as in Hybrid_4.

$\underline{\mathsf{Hybrid}_6}$: In this hybrid, we replace the correlation sampling of values of a pair of honest parties with PCG expansions (cf. case (1) of Hybrid_5). For example, instead of sampling $((\mathbf{a}_i,\mathbf{c}^{\mathsf{VOLE}}_{i,\ell,0}),\cdot)\in\mathcal{Y}_{\mathsf{VOLE}}(1^\lambda,(\rho_a^{(i)},\mathsf{sk}_\ell),[N])$, party P_i computes $(\mathbf{a}_i,\mathbf{c}^{\mathsf{VOLE}}_{i,\ell,0})=\mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(0,\mathsf{k}^{\mathsf{VOLE}}_{i,\ell,0})$. The same change is applied to all VOLE and OLE correlations.

Indistinguishability can be shown via a series of reductions to the pseudorandom $\mathcal{Y}_{\mathsf{VOLE}}$- and $\mathcal{Y}_{\mathsf{OLE}}$-correlated output property of the PCGs. In more detail, we construct a sequence of hybrid experiments where only a single correlation sampling is replaced by a PCG expansion. Then, in the reduction to the pseudorandom correlated output property, in case the challenge bit is 0, the reduction simulates the hybrid where the output is sampled from the correlation, and in case the challenge bit is 1, the output is the PCG expansion. A distinguisher between any pair of hybrid experiments in the sequence helps to construct a successful adversary against the pseudorandom correlated output property. We conclude that Hybrid_5 and Hybrid_6 are indistinguishable under the assumption of reusable PCGs.

$\underline{\mathsf{Hybrid}_7}$: Finally, we replace the reverse-sampling in case (2) of Hybrid_5 with the corresponding PCG expansion. For instance, instead of computing

$$(\mathbf{a}_i,\mathbf{c}^{\mathsf{VOLE}}_{i,j,0})\leftarrow\mathsf{RSample}_{\mathsf{VOLE}}(1^\lambda,(\rho_a^{(i)},\mathsf{sk}_j),0,(\mathsf{sk}_j,\mathbf{c}^{\mathsf{VOLE}}_{i,j,1}),[N])$$

the honest party computes

$$(\mathbf{a}_i,\mathbf{c}^{\mathsf{VOLE}}_{i,j,0})=\mathsf{PCG}_{\mathsf{VOLE}}.\mathsf{Expand}(0,\mathsf{k}^{\mathsf{VOLE}}_{i,j,0}).$$

The same change is applied for all other reverse-sampling algorithms.

Analog to the indistinguishability between Hybrid_5 and Hybrid_6, we can show indistinguishability between Hybrid_6 and Hybrid_7 via a sequence of hybrid experiments. In each hybrid one reverse sampling is replaced by the PCG expansion. Indistinguishability between adjacent hybrids is reduced to the security property of the PCG. Since the only change between two adjacent hybrids is the fact whether the correlation output of an honest party is reverse-sampled given the output of a corrupted party or taken as the PCG expansion, it is easy to see that a distinguisher between these hybrids can be used to construct a successful adversary against the security property.

We end up in Hybrid_7 where all correlation outputs and reverse-sampling outputs are replaced by PCG expansions. As this hybrid does not use any reverse-sampling anymore, we can get rid of the tuple function Tuple.

Now, Hybrid_7 is identical to the real-world execution which concludes the proof.

M Benchmarks of Basic Arithmetic Performance

We report the runtime of basic arithmetic operations in Table 1. The presented numbers might help the reader to assess the performance of system used for benchmarking and provides details for comparisons.

Table 1: Runtime of basic arithmetic operations in the BLS12_381 curve on our evaluation machine. The bit-size of the curve's group order p is 255. The error terms report standard deviation.

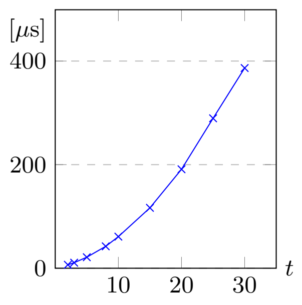
Operation	Time
$\mathbb{Z}_p$ addition	5.092 ns ±1.049 ns
$\mathbb{Z}_p$ multiplication	32.045 ns ±1.556 ns
$\mathbb{Z}_p$ inverse	2.713 μs ±101.973 ns
$\mathbb{G}_1$ addition	1.102 μs ±48.571 ns
$\mathbb{G}_2$ addition	3.668 μs ±96.867 ns
$\mathbb{G}_1$ scalar multiplication	279.146 μs ±14.763 μs
$\mathbb{G}_2$ scalar multiplication	0.952 ms ±0.04 μs
Pairing	2.403 ms ±56.976 μs

N Evaluation Considering [67]

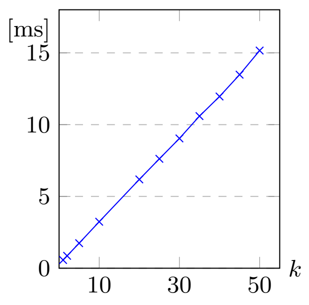
Concurrently to our work, Tessaro and Zhu [67] proposed and proved security of a more compact BBS+ signature scheme removing the nonce s, and hence, reducing the signature size by one element in $\mathbb{Z}_p$. The proposed extension translates to our protocol in a straight-forward way as follows. We do no longer need public parameter h_0. The preprocessing protocol does not generate the shares s_i or α_i. When answering a signing request, the servers compute A_i differently, i.e., $A_i := (g_1 \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell})^{a_i}$, and do not send s_i. The reconstruction of a signature ignores s and outputs the tuple (A, e). When verifying a signature, parties now check if $\mathsf{e}(A, y \cdot g_2^e) = \mathsf{e}(g_1 \cdot \prod_{\ell \in [k]} h_\ell^{m_\ell}, g_2)$. In the following we call the described protocol the *lean version of our protocol.*

For us, their optimization has the advantage of removing the necessity of the α values computed during the preprocessing and the computation of the g^{s_i} and g^s term in the signing and verification process. In order to quantify the benefits of this optimization, we have repeated the evaluation presented in Section 6 for the lean version of our protocol and report the changes here.

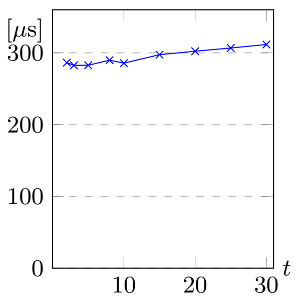
Online, Signing Request-Dependent Phase. For the online phase, we have implemented the lean version of the protocol and executed benchmarks. The scope of the implementation and the setup of our benchmarks remains unchanged. The results of our benchmarks are reported in Figure 11. The comparison to the non-threshold protocol, also optimized according to [67], is displayed in Figure 12. The size of signing requests does not change in the lean version of our protocol. The size of partial signatures sent by the servers reduces to $(2\lceil \log p \rceil + |\mathbb{G}_1|)$.



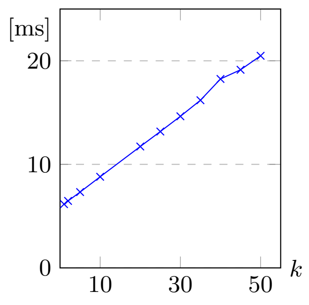
(a) Adapt (Server).



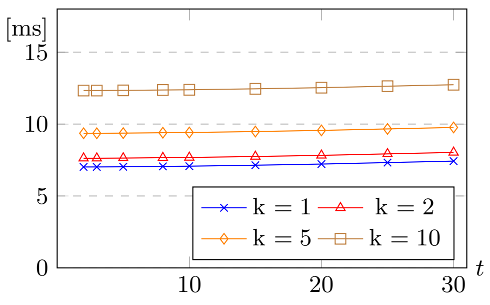
(b) Sign (Server).



(c) Reconstruct (Client).



(d) Verify (Client).



(e) Total.

Fig. 11: The runtime of individual phases (a)-(d) and the total online protocol (e) in the protocol version optimized according to [67]. The *Adapt* phase, describing Steps 5 and 6 of protocol π_{Prep}, and the *Reconstruct* phase, describing Step 3a of $\pi_{\mathsf{TBBS+}}$, depend on security threshold t. The *Sign* phase, describing Step 2 of $\pi_{\mathsf{TBBS+}}$, and the signature verification, describing Step 3b depend on the message array size k.

Offline, Signing Request-Independent Phase. For the offline phase, we derive the benchmarks for the lean version of our protocol from the original one. To this end, we have measured the execution time of the expansion steps that are removed by the lean version and deduct them from the total runtime. The results are displayed in Figure 14 and Figure 15. In the n-out-of-n setting of the lean version, each party performs four randomization and splits three polynomials. In the t-out-of-n setting, each party performs $2 + 3 \cdot (n - 1)$ randomizations and splits just as many polynomials. The time to extract one of the N field elements from a degree-N polynomial remains unchanged.

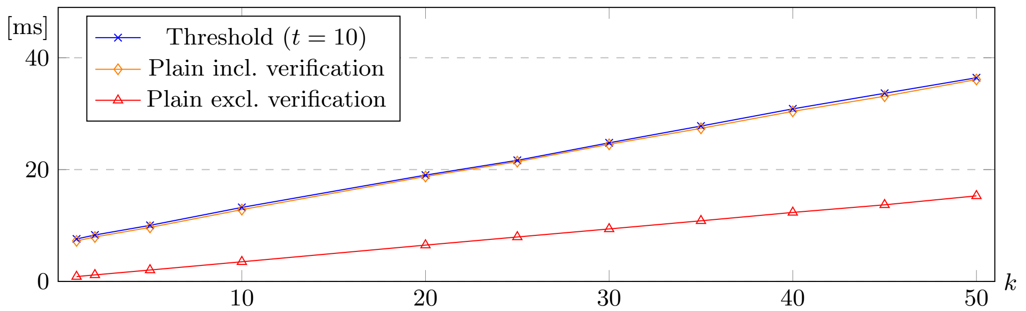


Fig. 12: The total runtime of the lean version of our online protocol in comparison to plain, non-threshold signing (also optimized according to [67]) with and without signature verification in dependence of the size of the message array k. As depicted in Figure 11e, the influence of the number of signers t is insignificant. We choose $t = 10$.

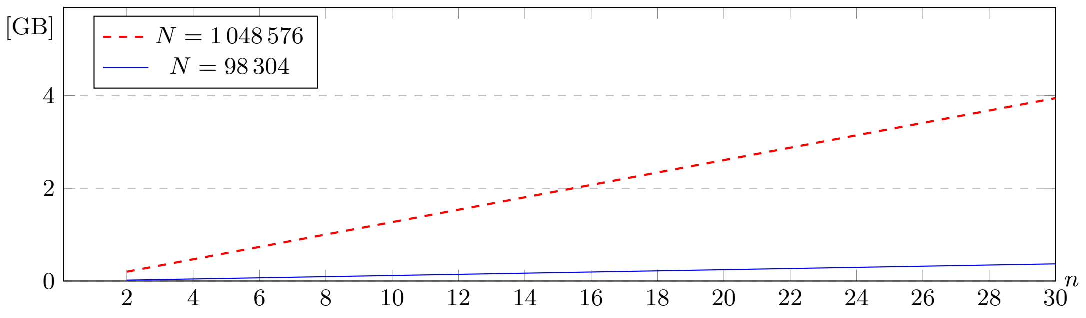


Fig. 13: Storage complexity of the preprocessing material in the lean version of our protocol required for $N \in \{98\,304, 1\,048\,576\}$ signatures depending on the number of servers n.

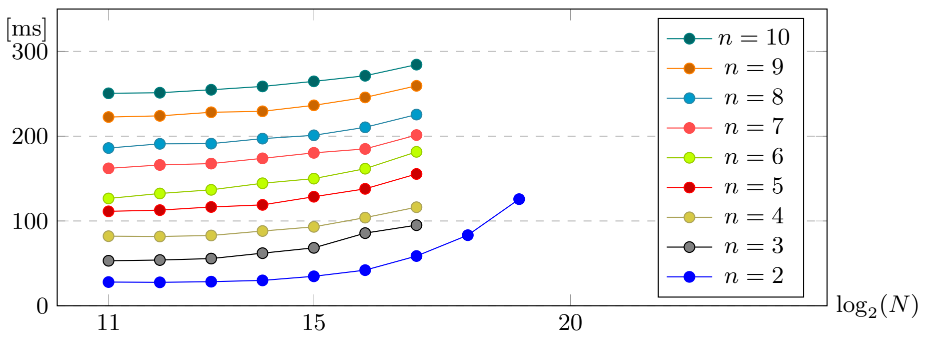


Fig. 14: Computation time in the lean version of our protocol of the seed expansion of all required PCGs in the n-out-of-n setting for different committee sizes ($n \in \{2, \ldots, 10\}$) dependent on the number of generated precomputation tuples N.

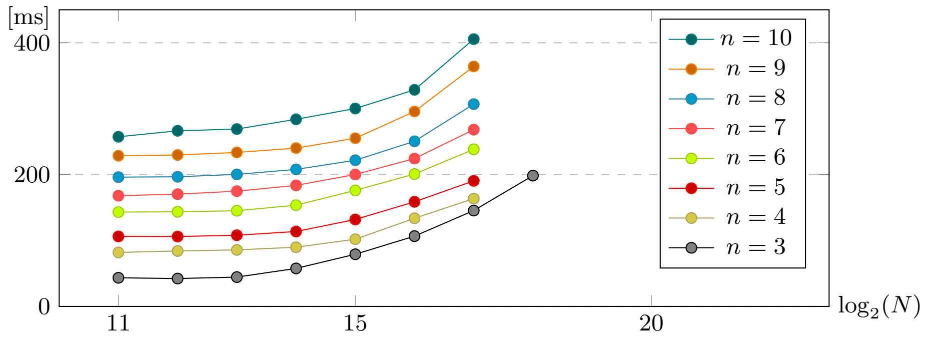


Fig. 15: Computation time in the lean version of our protocol of the seed expansion of all required PCGs in the t-out-of-n setting for different committee sizes ($n \in \{2, \ldots, 10\}$) dependent on the number of generated precomputation tuples N.

The communication complexity of a distributed PCG-based preprocessing protocol instantiating the offline, signing request-independent phase of the lean version of our protocol is dominated by a factor of

$$13(nc\tau)^2 \cdot (\log N + \log p) + 4n(c\tau)^2 \cdot \lambda \cdot \log N.$$

In case, the preprocessing decouples seed generation from seed evaluation, servers have to store seeds with a size of at most

$$\begin{aligned} &\log p + 2c\tau \cdot (\lceil \log p \rceil + \lceil \log N \rceil) \\ &+2 \cdot (n-1) \cdot c\tau \cdot (\lceil \log N \rceil \cdot (\lambda + 2) + \lambda + \lceil \log p \rceil) \\ &+2(n-1) \cdot (c\tau)^2 \cdot (\lceil \log 2N \rceil \cdot (\lambda + 2) + \lambda + \lceil \log p \rceil) \end{aligned}$$

bits. The expanded precomputation material occupies

$$\log p \cdot (1 + N \cdot (2 + 4 \cdot (n-1)))$$

bits of storage. In Figure 13, we report the concrete storage complexity of the preprocessing material of the lean version of our protocol when instantiating the with $N \in \{98\,304, 1\,048\,576\}$ and $p = 255$ according to the BLS12_381 curve used by our implementation.

The computation cost of the seed expansion is still dominated by the ones of the PCGs for OLE correlations. However, we do no longer need the OLE-generating PCGs for the cross terms $a_i \cdot s_j$, and $a_j \cdot s_i$. It follows that the computation complexity of the seed expansion in the lean version of our protocol is dominated by

$$2 \cdot (n-1) \cdot (4 + 2\lfloor \log(p/\lambda) \rfloor) \cdot N \cdot (ct)^2$$

PRG evaluations and $O(nc^2 N \log N)$ operations in $\mathbb{Z}_p$.