BEYOND MITIGATIONS:
Advancing Attack Surface Reduction and Analysis

Vom Fachbereich Informatik (FB 20)
an der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines Doktor-Ingenieurs
genehmigte Dissertation von:

MSc. Patrick Thomas Jauernig

Referenten:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (Erstreferent)
Prof. N. Asokan, Ph. D. (Zweitreferent)

Tag der Einreichung: 16. Oktober 2023
Tag der Disputation: 27. November 2023

TECHNISCHE
UNIVERSITÄT
DARMSTADT

System Security Lab
Fachbereich Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

Darmstadt 2023

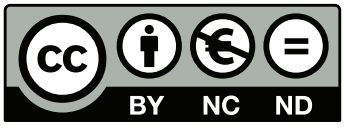
Patrick Thomas Jauernig:
Beyond Mitigations: Advancing Attack Surface Reduction and Analysis, © October 2023

Darmstadt, Technical University of Darmstadt
Day of dissertation defense: 27.11.2023
Dissertation publication at TUprints: 2024
URN of the dissertation: urn:nbn:de:tuda-tuprints-265292

DOCTORAL REFEREES:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (1st Doctoral Referee)
Prof. N. Asokan, Ph. D. (2nd Doctoral Referee)

FURTHER DOCTORAL COMMISSION MEMBERS:
Prof. Dr. Carsten Binnig
Prof. Dr. Sebastian Faust
Prof. Dr. Marco Zimmerling

Veröffentlichung unter CC-BY-NC-ND 4.0 International
Namensnennung, nicht kommerziell, keine Bearbeitung
https://creativecommons.org/licenses/



Erklärung gemäß §9 der Promotionsordnung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwednung der angegebenen Quellen und Hilfsmittel verfasst 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.

Darmstadt, Germany, October 2023

Patrick Thomas Jauernig

Abstract

In recent decades, we have witnessed an arms race between software attacks and defenses. This ongoing battle has seen modern computer systems incorporating a multitude of defenses, working collaboratively to shield sensitive applications and data from malicious attacks. Despite growing layers of security measures, vulnerabilities persist, often circumventing the most advanced safeguards and putting entire systems at risk. But how can we end this relentless cycle of attack and defense? One crucial aspect to systematically tackle the problem at hand is *attack surface reduction*, i.e., reducing the code that 1) is reachable by an attacker and 2) can also reach sensitive information. Attack surface reduction is not only applicable to code within an application but also extends to the broader software stack, including libraries and the operating system, which are inherently trusted components, often referred to as the *Trusted Computing Base* or TCB. Another crucial element is *attack surface analysis*, which assesses how vulnerable a program is. This analysis plays a pivotal role in uncovering vulnerabilities across the entire software stack, thereby bolstering the security of critical software components like the Trusted Computing Base (TCB). Although attack surface analysis is a well-established concept, recent advances, particularly in the realm of fuzzing, have begun to pave the way for its gradual adoption by the industry. Nonetheless, numerous challenges within this field still must be addressed to make it an integral part of the industry's software development process.

In this dissertation, we design, implement, and evaluate 1) novel attack surface reduction architectures using in-process isolation and enclaves, 2) protocols using these architectures as powerful primitives, and 3) an algorithmic improvement to fuzzing for attack surface analysis.

Secure In-Process Compartments. In-process isolation is an important building block for attack surface reduction within an application by segregating regular and sensitive data. Previous approaches primarily focused on broadly applicable isolation primitives, which allow developers to compartmentalize their applications at the cost of significant hardware and performance overheads.
In IMIX, we propose an instruction set extension for in-process memory isolation that allows high-frequency domain switching. IMIX is a minimally invasive approach to reduce the attack surface for highly sensitive data such that this data can only be accessed by the dedicated part of the application code. In contrast to state-of-the-art randomization schemes, IMIX safeguards the run-time defenses' metadata in a process deterministically. We implemented a prototype of IMIX to protect the metadata of the Code Pointer Integrity scheme with practical performance.

Flexible Enclaves for Application-driven Security. While in-process isolation can create different security domains within an application, Trusted Execution Environments (TEEs) allow the entire sensitive application to run within an isolated compartment (a so-called enclave) without trusting the operating system or hypervisor. Traditionally, these enclave architectures only provide a single enclave type per platform, which forces developers to adapt the application to the enclave's execution environment. This either increases the attack surface unnecessarily or restricts the sensitive app's functionality.
In CURE, we introduce the first TEE for flexible attack surface reduction based on the requirements of the sensitive application. CURE leverages system bus filtering to create different types of enclaves, enabling flexible per-enclave resource assignments for peripherals and even DMA devices. We prototype CURE for the open RISC-V architecture and evaluate its performance overhead and the hardware area overhead on an FPGA- and simulator-based setup.

Enclaves as Security Primitives in Protocols. While attack surface reduction with TEEs is a cornerstone of modern software protection, TEEs can also be leveraged as a security primitive to create powerful and efficient protocols. In this dissertation, we present TEE-based protocols for off-chain smart contract execution, and machine learning model protection.

Originally envisioned as the world computer, the smart contract ecosystem of Ethereum and other blockchains still largely consists of simple token manager contracts. As blockchain interactions are costly and rather slow, a crucial research area in this field is speeding up smart contract execution by performing the computation off-chain. Pure protocol-based works in this area only work for simple coin transfers, require frequent blockchain interactions, depend on collateral, or only marginally improve execution speed. TEEs proved to be a viable way to alleviate these shortcomings, as a single TEE can already give correctness guarantees for smart contract execution. However, the missing availability guarantees of TEEs prevents prior approaches from guaranteeing the complete execution of the smart contract. With POSE, we design a novel TEE-based off-chain smart contract execution protocol that is the first to provide strong liveness guarantees, while achieving private state without relying on collateral. We designed and implemented a prototype for Ethereum based on Arm TrustZone. We show that POSE is practical by evaluating numerous smart contracts, e.g., federated learning and Poker, and by measuring the time to execute individual steps of the protocol.

Another megatrend in computing is machine learning, especially artificial intelligence (AI), which is evolving quickly. As products increasingly rely on AI, machine learning models have become a high-value intellectual property for companies. Meanwhile, on the end-user device, privacy-sensitive user information is used to give personalized answers. Still, previous approaches for secure inference only addressed one of these aspects, incurred impractical requirements on the model, or had performance limitations. In OMG, we design a TEE-based protocol to protect both the machine learning model and the user's privacy. OMG leverages user-space enclaves to prevent model stealing attacks and protect sensitive user inputs. We implemented an offline wake word detection based on the TensorFlow lite for microcontrollers framework and the SANCTUARY enclave

architecture. We show that the machine learning inference protected within the strongly isolated exhibits unchanged accuracy and native inference speed.

Attack Surface Analysis with Fuzzing. Protecting blockchain and machine learning execution facilities with TEEs is effective and efficient. Yet, enclave code—same as regular applications—may contain vulnerabilities. Also, other components of the TCB may contain vulnerabilities. Hence, attack surface analysis is an essential cornerstone for modern security. In recent years, dynamic testing in the form of fuzzing has become increasingly popular. Fuzzers repeatedly execute a target, which can be software or even hardware, with random inputs and monitor the target for misbehavior or crashes. Fuzzing, in contrast to cumbersome manual approaches like unit testing, can test thousands of cases per second—from randomly generated inputs up to highly structured inputs, e.g., based on formal grammars. While initial research focused on technical advances to speed up the process and make targets fuzzable, current work also aims to optimize the fuzzer's internal algorithms to increase efficiency.
Lastly, in our work DARWIN, we propose a novel mutation scheduler for mutational fuzzing, designed for efficient and optimal selection of mutation operators with minimal performance impact on the fuzzing process. Further, DARWIN is straightforward to integrate into existing fuzzers and does not expose any target-dependent parameters. Thus, DARWIN can be integrated into most modern fuzzers. We show that DARWIN significantly improves time to coverage/bug while uncovering a completely novel bug in the extensively tested `objcopy`, which persisted for more than two decades.

Zusammenfassung

In den letzten Jahrzehnten haben wir ein Wettrüsten zwischen Software-Angriffen und Abwehrmaßnahmen erlebt. Dieser ständige Kampf hat dazu geführt, dass moderne Computersysteme eine Vielzahl von Schutzmaßnahmen enthalten, die zusammenarbeiten, um sensible Anwendungen und Daten vor bösartigen Angriffen zu schützen. Trotz der zunehmenden Zahl von Sicherheitsmaßnahmen gibt es nach wie vor Schwachstellen, die oft auch die fortschrittlichsten Schutzmaßnahmen umgehen und ganze Systeme gefährden.

Aber wie können wir diesen unerbittlichen Kreislauf von Angriff und Verteidigung beenden? Ein entscheidender Aspekt, um das Problem systematisch anzugehen, ist die Reduzierung der Angriffsfläche (*Attack Surface Reduction*), d. h. die Reduzierung des Codes, der 1) für einen Angreifer erreichbar ist und 2) auch sensible Informationen erreichen kann.

Die Reduzierung der Angriffsfläche gilt nicht nur für den Code innerhalb einer Anwendung, sondern erstreckt sich auch auf den breiteren Software-Stack, einschließlich der Bibliotheken und des Betriebssystems, bei denen es sich um inhärent vertrauenswürdige Komponenten handelt, die oft als Trusted Computing Base (TCB) bezeichnet werden. Ein weiteres wichtiges Element ist die Analyse der Angriffsoberfläche (*Attack Surface Analysis*), mit der die Anfälligkeit eines Programms bewertet wird. Diese Analyse spielt eine entscheidende Rolle bei der Aufdeckung von Schwachstellen im gesamten Software-Stack, wodurch auch die Sicherheit kritischer Softwarekomponenten wie der TCB erhöht wird. Obwohl die Analyse der Angriffsoberfläche ein etabliertes Konzept ist, haben die jüngsten Fortschritte, insbesondere im Fuzzing-Bereich, den Weg für ihre allmähliche Integration durch die Industrie geebnet. Nichtsdestotrotz müssen noch zahlreiche Herausforderungen in diesem Bereich bewältigt werden, um sie zu einem integralen Bestandteil des Softwareentwicklungsprozesses der Industrie zu machen.

In dieser Dissertation entwerfen, implementieren und evaluieren wir 1) neuartige Architekturen zur Reduzierung der Angriffsfläche durch in-process Isolation und Enclaves, 2) Protokolle, die diese Architekturen als leistungsfähige Primitive nutzen, und 3) eine algorithmische Verbesserung des Fuzzing zur Analyse der Angriffsfläche.

Secure In-Process Compartments. Die in-process Isolation ist ein wichtiger Baustein für die Reduzierung der Angriffsfläche innerhalb einer Anwendung. Hierbei werden reguläre und sensible Daten voneinander getrennt. Bisherige Ansätze konzentrierten sich

in erster Linie auf breit anwendbare Isolationsprimitiven, die es Entwicklern ermöglichen, ihre Anwendungen auf Kosten eines erheblichen Hardware- und Leistungs-Overheads zu isolieren.
In IMIX schlagen wir eine Befehlssatzerweiterung für die in-process Isolation vor, die ein hochfrequentes Domain Switching ermöglicht. IMIX ist ein minimal-invasiver Ansatz zur Reduzierung der Angriffsfläche für hochsensible Daten, so dass diese Daten nur vom dedizierten Teil des Anwendungscodes abgerufen werden können. Im Gegensatz zu modernen Randomization-Ansätzen sichert IMIX die Metadaten der Laufzeitverteidigung in einem Prozess deterministisch ab. In unserer Arbeit haben wir einen Prototyp von IMIX implementiert, um die Metadaten des Code Pointer Integrity Schemas mit minimalem Performance-Overhead zu schützen.

Flexible Enclaves for Application-driven Security. Während die in-process Isolation verschiedene Sicherheitsdomänen innerhalb einer Anwendung schaffen kann, ermöglicht TEEs die Ausführung der gesamten sensiblen Anwendung innerhalb eines isolierten Bereichs (einer so genannten Enclave), die dann weder Betriebssystem noch Hypervisor vertrauen muss. Traditionell bieten diese Enclave-Architekturen nur einen einzigen Enclave-Typ pro Plattform, was die Entwickler zwingt, die Anwendung an die Ausführungsumgebung der Enclave anzupassen. Dies vergrößert entweder die Angriffsfläche unnötig oder schränkt die Funktionalität der sensiblen Anwendung ein.
In CURE stellen wir das erste TEE zur flexiblen Reduzierung der Angriffsfläche auf der Grundlage der Anforderungen der sensiblen Anwendung vor. CURE nutzt ein System-Bus-Filtering, um verschiedene Typen von Enclaves zu kreieren, die eine flexible Ressourcenzuweisung pro Enclave für Peripheriegeräte und sogar DMA-Geräte ermöglichen. Wir implementieren einen Prototyp von CURE für die offene RISC-V-Architektur und evaluieren den Leistungs- und Hardwareflächen-Overhead auf einem FPGA- und Simulator-basierten Setup.

Enclaves as Security Primitives in Protocols. Während die Reduzierung der Angriffsfläche mit TEEs ein Eckpfeiler des modernen Softwareschutzes ist, können TEEs auch als Sicherheitsprimitiv genutzt werden, um leistungsstarke und effiziente Protokolle zu erstellen. In dieser Dissertation stellen wir TEE-basierte Protokolle für die Ausführung von Smart Contracts außerhalb der Blockchain und zum Schutz von Machine-Learning-Modellen vor.

Ursprünglich als Weltcomputer konzipiert, besteht das Smart-Contract-Ökosystem von Ethereum und anderen Blockchains immer noch weitgehend aus einfachen Token-Manager-Contracts. Da Blockchain-Interaktionen kostspielig und eher langsam sind, besteht ein wichtiger Forschungsbereich in diesem Bereich darin, die Ausführung von Smart Contracts zu beschleunigen, indem die Berechnungen außerhalb der Blockchain durchgeführt werden. Reine protokollbasierte Arbeiten in diesem Bereich funktionieren nur für einfache Coin Transfers, erfordern häufige Blockchain-Interaktionen, sind von

einem Collateral abhängig oder verbessern die Ausführungsgeschwindigkeit nur geringfügig. TEEs haben sich als praktikabler Weg erwiesen, diese Mängel zu beheben, da ein einziges TEE bereits Korrektheitsgarantien für die Ausführung von Smart Contracts geben kann. Allerdings verhindern die fehlenden Availability-Garantien von TEEs, dass frühere Ansätze die vollständige Ausführung des Smart Contracts garantieren können. Mit POSE entwerfen wir ein neuartiges, auf TEEs basierendes Protokoll zur Ausführung von Smart Contracts außerhalb der Blockhain, das als erstes starke Liveness-Garantien bietet und gleichzeitig Private State bietet, ohne sich auf ein Collateral zu verlassen. Wir haben einen auf Arm TrustZone basierenden Prototyp für Ethereum entworfen und implementiert. Mit diesem zeigen wir, dass POSE praktikabel ist, indem wir zahlreiche Smart Contracts evaluieren, z. B. Federated Learning und Poker, und indem wir die Zeit zur Ausführung einzelner Schritte des Protokolls messen.

Ein weiterer Megatrend im Bereich der Datenverarbeitung ist das maschinelle Lernen (ML), insbesondere die künstliche Intelligenz (KI), die sich rasch weiterentwickelt. Da sich Produkte zunehmend auf KI stützen, sind die Modelle des maschinellen Lernens für die Unternehmen zu einem wertvollen geistigen Eigentum geworden. Gleichzeitig werden auf den Nutzer-Geräten datenschutzrelevante Benutzerinformationen verwendet, um personalisierte Antworten zu geben. Bisherige Ansätze für eine sichere Inference befassten sich jedoch nur mit einem dieser Aspekte, stellten unpraktische Anforderungen an das Modell oder wiesen Leistungseinschränkungen auf.
In OMG entwickeln wir ein TEE-basiertes Protokoll, das sowohl das ML Modell als auch die Privatsphäre des Benutzers schützt. OMG nutzt User-Space Enclaves um Model-Stealing-Angriffe zu verhindern und sensible Nutzereingaben zu schützen. Wir haben eine Offline-Wakeword-Erkennung implementiert, die auf dem TensorFlow Lite for Microcontrollers Framework und der SANCTUARY Enclave Architektur basiert. Wir zeigen, dass die ML Inference, die innerhalb der stark isolierten Enclave geschützt ist, eine unveränderte Genauigkeit und native Inference-Geschwindigkeit aufweist.

Attack Surface Analysis with Fuzzing. Der Schutz von Blockchain- und Machine-Learning-Ausführungsumgebungen mit TEEs ist effektiv und effizient. Dennoch kann der Enclave-Ccode - ebenso wie reguläre Anwendungen - Schwachstellen enthalten. Auch andere Komponenten der TCB können Schwachstellen enthalten. Daher ist die Analyse der Angriffsoberfläche ein wesentlicher Eckpfeiler für moderne Sicherheit. In den letzten Jahren wurde das dynamische Testen in Form von Fuzzing immer beliebter. Fuzzer führen ein Target, bei dem es sich um Software oder sogar Hardware handeln kann, wiederholt mit zufälligen Eingaben aus und überwachen das Target auf Fehlverhalten oder Abstürze. Im Gegensatz zu mühsamen manuellen Ansätzen wie Unit-Tests, kann Fuzzing Tausende von Fällen pro Sekunde testen - von zufällig generierten Eingaben bis hin zu stark strukturierten Eingaben, z. B. auf der Grundlage formaler Grammatiken. Während sich die anfängliche Forschung auf technische Fortschritte konzentrierte um den Prozess zu beschleunigen und Ziele fuzzbar zu machen, zielen neuere Ansätze auch darauf ab, die internen Algorithmen des Fuzzers zu optimieren, um die Effizienz zu

erhöhen.
Schließlich schlagen wir in unserer Arbeit, DARWIN, einen neuartigen Mutation-Scheduler für Mutations-basiertes Fuzzing vor, der für eine effiziente und optimale Auswahl von Mutationsoperatoren mit minimalen Leistungseinbußen auf den Fuzzing-Prozess entwickelt wurde. Darüber hinaus ist DARWIN einfach in bestehende Fuzzer zu integrieren und führt keine neuen Target-abhängigen Parameter ein. Daher kann DARWIN in die meisten modernen Fuzzern einfach integriert werden. Wir zeigen, dass DARWIN die Zeit bis zur Entdeckung eines Bugs erheblich verkürzt und gleichzeitig einen völlig neuen Bug in dem ausgiebig getesteten `objcopy` aufdeckt, der mehr als zwei Jahrzehnte lang bestehen blieb.

Contributions

This dissertation is based on five scientific publications, which are the result of excellent collaboration with highly-experienced researchers and students, which I all thank for their valuable contributions. In the following, I outline my contributions to each publication included in this dissertation.

Chapter 2 is based on IMIX [70], a joint work with Tommaso Frassetto, Christopher Liebchen, and Ahmad-Reza Sadeghi. For IMIX, the design was a collaboration between Christopher Liebchen and me. I was the main author of the implementation, and integrated our solution in existing code for a use case. I also performed the performance evaluation. Tommaso contributed to the paper writing. This paper was developed in parallel with my master thesis [99]. While the thesis targets a simplified memory corruption defense as a use case, we extend the implementation to target the Code Pointer Integrity defense. Further, the paper contains newly implemented approximations for related hardware primitives and a new security analysis.

Chapter 3 is based on CURE [13], a joint work with Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Ghada Dessouky, Emmanuel Stapf and myself contributed to the discussions on the design and implementation that resulted in this publication. Emmanuel Stapf conceived the main idea and led the work, and supervised the M.Sc. thesis of co-author Matthias Klimmek whose work focused on the software stack implementation of the CURE architecture and its evaluation. I focused on the implementation of CURE's modifications at the processor and the software stack evaluation. Ghada Dessouky focused on the design, implementation, and evaluation of the cache partitioning for CURE. Emmanuel Stapf focused on the design of CURE's modifications at the processor, the design and implementation of CURE's access control mechanisms at the system bus, and led the evaluation. Raad Bahmani and Ferdinand Brasser contributed to the paper writing.

Chapter 4 is based on a joint works with Tommaso Frassetto, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi, and with Sebastian P. Bayerl, Tommaso Frassetto, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. For POSE [68], David Koisser, David Kretzler, Benjamin Schlosser, Tommaso Frassetto, and myself contributed to the discussions on the design that resulted in this publication. In detail, David Koisser, David Kretzler, and Benjamin Schlosser focused on the protocol, while I was responsible for the design of the off-chain execution environment. David Kretzler implemented the

on-chain manager contract, David Koisser the blockchain interaction, and I focused on the implementation of the system side of POSE and the corresponding evaluation of the off-chain smart contract execution. Benjamin Schlosser and David Kretzler proved the security properties of the protocol, while David Koisser and I analyzed the system's security guarantees. For OMG [17], Sebastian P. Bayerl, Tommaso Frassetto, Emmanuel Stapf, Christian Weinert, and myself contributed to the discussions on the design and implementation that resulted in this publication. I focused on the design of the communication protocol used by the SANCTUARY enclave, the user and the vendor of the machine learning model. Tommaso Frassetto focused on porting the TensorFlow Lite machine learning framework to SANCTUARY. Sebastian P. Bayerl focused on preparing the machine learning model and test data used during the evaluation of SANCTUARY. Emmanuel Stapf focused on the implementation of the keyword recognition algorithm in a SANCTUARY enclave and its evaluation. Christian Weinert contributed to the paper writing.

Chapter 6 is based on DARWIN [97], a joint work with Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. For DARWIN, Domagoj Jakobovic, Stjepan Picek, and myself contributed to the design of the mutation scheduler. I was the lead author of the design. Domagoj Jakobovic and I implemented the mutation scheduling algorithm, while I performed the evaluation of the prototype. Emmanuel Stapf contributed to the paper writing.

Acknowledgements

This journey, now drawing to a close, has been shaped significantly by numerous people to whom I owe my deepest gratitude.

Foremost among them is my advisor, Prof. Ahmad-Reza Sadeghi. His unwavering commitment to excellence and his ability to inspire the best in me have been instrumental. His guidance granted me the freedom to explore my research while equipping me with the skills to present our findings in an understandable and concise way, not only in academia but also in numerous industry projects.

I also owe a debt of gratitude to my colleagues, many of whom have become dear friends over the years. Special thanks are due to Ferdinand and Emmanuel, who made my dream of starting a company a reality. I am equally grateful to Tommaso and David, my constant companions in both scientific and personal discussions.

My research projects would not have been possible without the collaboration and insights of my co-authors. Thus, my thanks go to all my colleagues from the System Security Lab and external collaborators. Additionally, I am thankful to Prof. Asokan for graciously accepting the role of my second doctoral referee. My appreciation also goes to the rest of the defense committee members – Prof. Carsten Binnig, Prof. Sebastian Faust, and Prof. Marco Zimmerling – for their engaging and thought-provoking discussions during my dissertation defense.

I must also extend my heartfelt gratitude to my friends, whose constant presence and support played a crucial role in my journey. Their understanding and compassion provided solace during the most challenging times, making this journey more bearable.

Last but certainly not least, I must express my heartfelt thanks to my family, whose unwavering support made it possible for me to pursue this academic path. A special thanks goes to Maren, who endured many missed evenings, weekends, and vacation days to support my journey. Your sacrifice and support have been crucial in bringing this chapter of my life to a successful conclusion.

Contents

1

Introduction

Over the last decades, computers evolved from specialized experimental equipment to the cornerstone of our digital society. New chip designs and vastly improved manufacturing capabilities allowed computer systems to be integrated into every aspect of our lives. To meet the demands of this widespread integration, software development has been in a constant state of evolution, introducing new and complex features at an unprecedented rate. Consequently, code bases have ballooned in size—the Linux kernel, for instance, has roughly doubled its size over the last ten years [152, 113], and it now comprises over 35 million lines of code [183]. However, application software like web browsers has already overtaken Linux [92]. This rapid expansion in software features contributed to an explosion of reported vulnerabilities [219], including catastrophic security flaws [134, 182] that deeply shattered the trust in the computer software.

A significant contributor to today's software security challenges is the prevalence of historically grown, legacy code, often written in memory-unsafe languages such as C and C++. These memory-unsafe languages put the security burden entirely on the developer, as they 1) require active lifecycle management of objects in memory and 2) allow the developer to manipulate references to memory (pointer arithmetic) liberally. Unsurprisingly, these practices have led to a surge in memory-corruption vulnerabilities: memory safety violations that can be exploited to hijack the control flow of an application. Alarmingly, despite the availability of modern, memory-safe programming languages like Rust [105], new memory-unsafe code continues to be written [175], sustaining the cycle of vulnerability creation. Thus, programming errors that lead to memory-corruption vulnerabilities remain a persistent and critical issue in the software landscape [133].

Fortunately, both academic research and industry have not been idle; they have developed a plethora of defenses, commonly known as *mitigations*, aimed at curtailing the impact of these vulnerabilities. Broadly, these mitigations can be classified into two categories: integrity-based and randomization-based approaches. The former ensures the integrity of the control flow, e.g., by comparing each control-flow transition against precomputed ground truths [3, 31, 69], or by ensuring the integrity of high-value data like code pointers and other program structures [112, 32, 216]. Recently, these mitigations have been implemented as hardware primitives to be used by software [120, 122, 121, 168], which further reduces the mitigations' attack surface towards a software adversary.

However, these methods often hinge on dedicated metadata for ensuring data integrity, e.g., to compare a potentially tampered code pointer to a backup in the metadata structure. This creates a potential weak point: if this metadata is compromised, the mitigation can be entirely subverted [36, 65, 112].

In contrast, randomization-based approaches leverage the high entropy of virtual address spaces to shuffle the memory layout [23, 47, 27, 50]. Thus, to successfully exploit a vulnerability, an adversary needs to reverse this randomization to find the address of valuable data, e.g., using an information leak [172]. Indeed, many deployed mitigations have been bypassed [172, 46, 65, 166, 212, 12, 18, 45], making memory-corruption vulnerabilities remain an important attack vector in practice.

To finally address the problem at hand systematically, two factors are critical: 1) clear attack surface reduction, where a large proportion of the legacy software's dependencies (e.g., libraries or its execution environment) cannot affect the security of sensitive code anymore, and 2) more advanced ways to analyze the attack surface to assess the remaining risk for vulnerabilities within the code.

In this dissertation, we not only explore advances in attack surface reduction and analysis techniques but also how attack surface reduction primitives can help to construct efficient and powerful protocols.

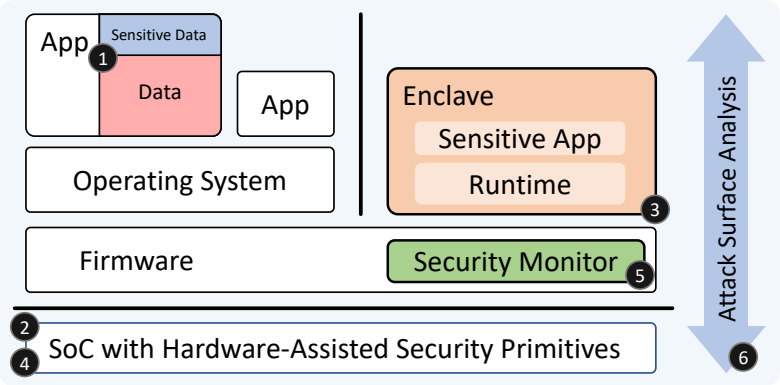


Figure 1: High-level overview of the complex software stack on a modern system.

1.1 Attack Surface Reduction

Attack surface refers to all the entry points that the software exposes which can potentially be leveraged by an adversary to attack a target. However, knowing which part of the software belongs to the attack surface is highly challenging as it requires a deep understanding of the software architecture, design, and functionality. In addition, the software, and hence, also the attack surface, is not static but can change over time due to updates. Hence, defensive measures need to concentrate on the highly sensitive parts of

the software. Guarding these sensitive parts, such that an adversary cannot reach them, is what compartmentalization and trusted execution environments address.

In-Process Compartmentalization

Memory corruption attacks remain a significant issue in computer security, despite the use of various preventive measures such as safe programming languages, static and dynamic analysis, and run-time defenses (also called mitigations). Especially the latter has been subject to extensive research. These run-time defenses can be broadly classified into randomization-based and integrity-based approaches. Randomization-based approaches seek to obscure the location of critical code or data blocks in an application by randomly rearranging or hiding them within the virtual address space of the application. While this can be effective in some cases, it is ultimately a probabilistic defense that can be bypassed with enough tries or if an attacker can discover information leaks that reveal the location of these blocks [172].

Instead, integrity-based approaches aim to enforce an application's correct behavior, e.g., by verifying the integrity of code pointers or checking for valid control-flow transitions. These approaches are often more robust against information leaks, yet may require more coarse-grained checks to maintain an acceptable performance overhead. However, most integrity-based techniques rely on metadata to ensure the integrity of code or data blocks. As memory-corruption adversaries are commonly assumed to have full read and write access to the application's memory (except code pages which are nowadays protected by execute-only permissions), the metadata is also at risk. Once this metadata is corrupted, the defense is broken entirely.

Protecting this metadata is challenging, as integrity-based approaches must frequently access this data. While hiding the metadata in virtual memory is fast, information leaks or brute-force attacks endanger the security of the integrity scheme [65]. As shown in Figure 1, secure compartments ❶ can prevent illegal access to this metadata deterministically (opposed to probabilistic protection of randomization), either as pure software-based approaches [193, 167, 23, 108], or leveraging hardware primitives. Indeed, several hardware-assisted compartments ❷ are already available in practice, most notably Intel MPX [144] for bounds checking, Arm Memory Domains [217] (not available anymore in recent architectures) and Memory Protection Keys (MPK) [104] for creating memory domains with dedicated access rights. However, these approaches do not support frequent switching between the different memory domains they create [108], e.g., between regular code and mitigation code. Recently, Intel introduced Control-flow Enforcement Technology (CET) [169], which aims to address this by adding hardware support for indirect branch tracking (to implement CFI checks) and a shadow stack, which adds hardware instructions to create and verify backups of the return addresses on the stack. However, this mechanism is only designed for shadow stacks. Hence, CET is not flexible enough to safeguard other mitigations.

Trusted Execution Environments

While in-process compartmentalization helps to better protect individual applications from compromises, each application must be protected individually. If only one application is compromised, the large attack surface of the operating system (OS) and third-party device drivers create a significant risk for other applications. Further, e.g., in cloud scenarios, customers might not trust the cloud vendor's hypervisor and fear direct attacks or side-channel leakage to other tenants.

To address this growing problem, sensitive applications' attack surface must be reduced tremendously. One promising technology for achieving this is Trusted Execution Environment (TEE), also called enclave architectures. TEEs provide strongly isolated compartments ❸, known as enclaves, which protect sensitive applications, i.e., sensitive parts of an application, from other applications and privileged software such as the OS and the hypervisor. This isolation is enforced by hardware primitives ❹ configured by a small trusted software ❺, often called the security monitor. TEEs offer a promising approach to enhance the security of sensitive applications by reducing their attack surface and providing more robust protection against malicious attacks. TEEs are available on many commercial CPUs already, e.g., Arm TrustZone-A [9] and TrustZone-M [210], Arm Confidential Compute Architecture [128], Intel Software Guard Extensions (SGX) [94, 48] and Trust Domain Extensions (TDX) [93], AMD Secure Encrypted Virtualization (SEV) [102, 6, 103, 7], or IBM Protected Execution Facility [86]. However, most of these architectures were already bypassed to leak secrets from enclaves [213, 190, 137, 191].

Enclave Applications. Enclaves are a simple and effective way to protect an application from the large attack surface of a regular system. They have been used for this purpose in various scenarios, such as databases, digital rights management, and software containers. In addition to their use as single-purpose high-security vaults, enclaves also serve as powerful security primitives that can be integrated into higher-level protocols to improve security, performance, and scalability. However, TEEs are not a foolproof technology and do neither guarantee availability for the service in an enclave, nor the confidentiality or timeliness for communication between enclaves and external services (e.g., network or filesystem). As a result, designing protocols for applications in scenarios such as blockchain is highly challenging.

1.2 Attack Surface Analysis

Enhancing the security of sensitive applications often involves the use of isolation mechanisms like enclaves to shield them from external threats. However, the isolation enforced by the enclave's enclave architecture is not a panacea; both the trusted software com-

ponents of the enclave architecture and the sensitive applications it hosts could contain vulnerabilities. Given these complexities, the dual strategies of attack surface reduction and attack surface analysis become even more critical. Attack surface reduction serves to minimize the avenues through which an adversary could compromise a system, thereby reducing the risk profile of both enclaves and the applications they safeguard. Conversely, attack surface analysis provides a vital complement by systematically identifying residual vulnerabilities and potential entry points ❸, including those within the enclave-protected applications themselves. Importantly, this exhaustive scrutiny needs to extend beyond applications to encompass all layers of the computational stack: enclaves, operating systems, hypervisors, firmware, security monitors, and even hardware.

While traditional approaches for attack surface analysis require extensive labor, e.g., in the form of manual writing of unit tests, modern approaches can reduce the effort tremendously by exhaustive automatic analysis. Automatic analysis has two main research directions: static analysis and dynamic analysis. Static analysis extends the compilation process to look for error-prone programming patterns or basic memory mismanagement. However, static analysis does not scale well: it operates on a symbolic representation of the code, requiring the evaluation of every branch under every possible value range within the input domain. Dynamic analysis, in contrast, executes the program directly to find bugs. Recently, fuzzing has emerged as a popular research direction in this area. Fuzzing tests the target program by repeatedly running it with randomly generated inputs. If the target program crashes, a bug has been found. While initially developed for regular applications [74], fuzzing is now used within all software privilege layers and even for testing hardware designs [188, 37].

1.3 Dissertation Outline

In the following, we give a brief outline of the remainder of this dissertation, whereby each chapter presents the respective publication in more detail and situates the publication within the broader context of relevant literature.

Chapter 2: We present IMIX, a novel lightweight intra-process isolation design where memory pages are tagged as security-sensitive, while a specialized instruction in the instruction set architecture (ISA) provides exclusive access to these pages. While IMIX can protect arbitrary sensitive data, IMIX is designed to efficiently safeguard the metadata associated with memory-corruption defenses.

Chapter 3: We introduce CURE, the first Trusted Execution Environment (TEE) architecture designed that provides multiple types of enclaves at once, offering a level of flexibility that allows adaptation to the unique requirements of sensitive applications. Additionally, CURE introduces fine-grained resource management capabilities to ensure that enclave resource demands are met without expanding the attack surface unnecessarily.

Chapter 4: We present POSE and OMG, protocols that harness the power of Trusted Execution Environments (TEEs) to improve security guarantees and performance for blockchain and AI applications. POSE, an off-chain smart contract execution protocol, offers strong liveness guarantees while ensuring private state without collateral requirements. Furthermore, POSE accelerates smart contract execution, enabling modern use cases such as machine learning integration.

Offline Model Guard (OMG), on the other hand, presents a secure and private machine learning approach, even in offline scenarios. OMG leverages TEEs to establish strict isolation between models and users, with additional support for hardware-based machine-learning accelerators.

Chapter 5: We introduce DARWIN, a lightweight mutation scheduler for fuzzing based on the Evolution Strategy algorithm. DARWIN optimizes the mutation-selection probability distribution throughout the fuzzing process, utilizing coverage feedback as a fitness function. In contrast to existing approaches, DARWIN improves coverage and bug-finding capabilities while avoiding the introduction of new per-target parameters that necessitate manual tuning.

2

Secure In-Process Compartments

As mentioned in Chapter 1, an adversary can exploit memory-corruption vulnerabilities to gain arbitrary code execution within an application at run time (commonly referred to as a *process*). This can be done by manipulating memory locations to either inject new code, reuse existing code, or perform so-called data-only attacks. Code-injection attacks add new, malicious code to the process, while code-reuse attacks recombine existing code of the process to perform arbitrary computations. Data-only attacks try to steer the control flow by modifying data variables, e.g., used to determine branches, within the existing control-flow graph. Research has shown that even data-only attacks can perform arbitrary computations [96].

Thus, code must be hardened against such attacks by integrating defenses to guard (control) data. The most straightforward attack, code injection, can effectively be prevented by enforcing that a memory page can either be writable or executable, but not both ($W \oplus X$). This prevents injected code from being executed and is indeed used in practice since 2004 [178]. Yet, code-reuse and data-only attacks are still possible, and hence, countless software defenses emerged to prevent these attacks, e.g., shadow stacks [32], Code Pointer Integrity (CPI) [112], or various forms of Control-Flow Integrity (CFI) [3, 46, 31, 69]. However, these defenses can be the target of an adversary themselves, especially as they need to store their own metadata in the same address space. This metadata is crucial for a working mitigation, and an adversary can manipulate this data to bypass a mitigation [65, 32]. While storing the metadata in another process or privilege layer (e.g., in the kernel) seems appealing, software defenses require frequent access to the metadata, e.g., for every code pointer load, and switching between these contexts is slow [108]. Hence, this metadata is usually hidden in the process's virtual address space by exploiting the larger entropy of today's 64-bit systems. Unfortunately, a single information leak (register spilling, brute-force guessing) has been shown to be enough to break this approach completely [172, 65].

Therefore, a deterministic isolation approach, as opposed to probabilistic randomization schemes, is needed. One important primitive that emerged to secure such metadata is intra-process isolation. Intra-process isolation creates an isolated compartment ("safe region") within the process that can only be accessed when in a specific context, e.g., when a return address is stored to/loaded from the shadow stack.

Intra-process isolation is a crucial building block for securing memory-corruption defenses. Designing such a scheme is challenging due to the wide-ranging requirements, namely, the approach must...

1. protect a safe region deterministically,
2. be easy to integrate into existing applications or memory-corruption defenses, ideally without changing the source code,
3. induce very little performance overhead (as this is on top of the defenses' performance overhead),
4. leverage an isolation primitive that supports frequent invocations, e.g., for every code pointer dereference.

2.1 Our Contributions

This thesis addresses these requirements on intra-process isolation with the following publication, which can be found in Appendix A.

[70] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 83–97. USENIX Association, 2018. CORE Rank A*. Appendix A.

With IMIX, we propose a novel lightweight intra-process isolation primitive that marks memory pages as security sensitive, which can only be accessed with our newly added instruction in the Instruction Set Architecture (ISA). This efficiently protects the metadata of memory-corruption defenses. Yet, the design of IMIX is not tailored towards a specific memory-corruption defense or ISA. In the following, we provide a more detailed summary of our approach.

IMIX. As shown in Figure 2, IMIX introduces a new permission bit to mark pages as isolated, i.e., belonging to the safe region. This permission bit is similar to the bit flags for read/write/execute. We further added support for IMIX in the kernel, which allows developers to mark a safe region using existing memory management functionality. If an application wants to leverage IMIX to guard a sensitive region, it first has to mark the region as sensitive. On Linux, this would be done with the `mmap` system call.

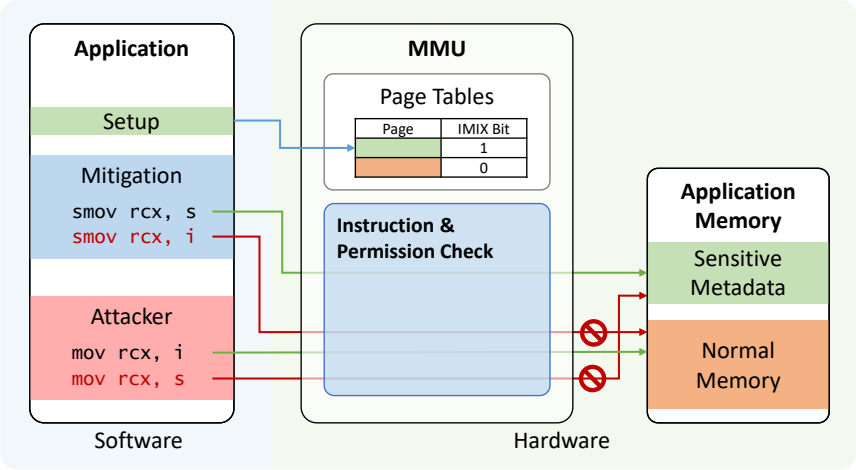


Figure 2: High-level overview of IMIX

Further, IMIX adds a new instruction to the ISA, `smov`, which is a pendant to the regular move instructions. This `smov` instruction encodes the security-sensitive context, such that only `smov` instructions can access the safe regions (but not the remaining memory, which prevents data-flow attacks on `smov` instructions originating from normal memory). Analogously, regular `mov` instructions can access all normal memory except safe regions. Hence, the application has to ensure that only code belonging to the sensitive region uses the `smov` instruction, while all other code should use a regular `mov` instruction. To simplify this, IMIX provides a compiler pass that can be leveraged by mitigations that want to protect their metadata, or by application developers themselves.

Implementation. We implemented a prototype of IMIX for x86-64. For every memory page, we maintain a flag in the Page Table Entry belonging to the page to indicate whether it belongs to the safe region. We further extended the Memory Management Unit of the CPU to check whether the current instruction accessing the memory was a `smov` instruction, and if the flag in the Page Table Entry is set. We implemented this CPU extension in the Windriver Simics simulation framework [1], which enables instrumentation on off-the-shelf Intel CPU designs.

We further extended the Linux kernel to integrate the IMIX permission bits in the memory management functions (e.g., `mmap`).

For application-side support, we implemented an LLVM-based compiler pass. The compiler pass can emit `smov` instructions by directly invoking IMIX library functions or by inserting source code annotations. We further implemented a use case based on CPI [112] and modified the code to use `smov` instructions for storing/loading metadata.

We evaluated CPI using the standard SPEC CPU2006 benchmark suite and compared its performance to plain CPI with direct memory access and randomization-based CPI, i.e., segment-offset-based access. The original memory-hiding-based CPI implementation induced a 4.24% performance overhead (geometric mean), while the IMIX variant induces a 3.99% performance overhead. We further implemented a CPI variant that uses an approximation [108] of Intel's MPK. This variant induced an overhead of 12.43%, showing MPK's shortcomings in high-frequency domain changes.

2.2 Related Work

Retrofitting Existing Isolation Features. Building an intra-process isolation primitive out of existing hardware features is appealing, as they are available on off-the-shelf hardware. However, approaches leveraging existing hardware functionality often need to work around a caveat of that specific feature—either due to different performance goals (e.g., not made for frequent domain switching) or due to non-optimal requirements (e.g., privileges) of the primitive. This results in additional software components like instruction filters or call gates that require protection themselves.

One research direction leverages the Memory Management Unit (MMU) to implement different views on memory. Software-fault Isolation (SFI) techniques [193, 167, 55, 23] reserve a part of the virtual address space for the safe region and apply offsets to allowed instructions automatically. However, they reduce the available virtual address space. Other SFI approaches leverage hardware-based bounds checking [35, 108] to avoid virtual address space reduction, but do not support frequent switching [144, 108]. Another approach is to use different virtual address spaces for different execution contexts (like for safe regions) [124], however, these techniques do not protect from overflows into the safe region [35]. IMIX checks permissions for `mov` and `smov` instructions such that neither can overflow into the other respective region while keeping the whole virtual address space available and enabling fast and frequent domain switching.

A different line of research uses protection keys for user space to design intra-process isolation schemes [217, 108, 189, 83, 149, 165, 207, 53]. Protection Keys for Userspace (PKU) are , e.g., available on recent Intel x86 processors, where the feature is called Intel Memory Protection Keys (MPK). While we will focus on MPK as a representative of general protection key schemes here, other implementations exist, e.g., on older Arm processors [217, 40]. MPK enables developers to assign memory pages to a set of memory domains. Subsequent memory accesses are only possible if the instruction executing is in an allowed domain. Entering this domain is done by setting a dedicated user space register. However, MPK is not designed for frequent switches: for each sensitive data access, the code needs to read the domain register, change the domain register, perform the access, and restore the domain register [108].

In addition, all unprivileged code can tamper with page permissions in the MPK register [189]. Hence, MPK instructions not originating from the intra-process isolation need to be filtered out, as, e.g., proposed in ERIM [189] and HODOR [83]. This 1) is dangerous as parsers are prone to memory-corruption errors themselves, 2) requires ensuring that another process cannot indirectly tamper with the registers through the kernel interfaces, and 3) requires that the filtering step is exhaustive as a single remaining MPK domain change instruction could be used as a primitive by an adversary to break the scheme. In fact, exhaustive filtering is nearly impossible on x86 due to unaligned instructions and has been circumvented [45, 200]. Eliminating the filtering step is not possible for MPK due to the direct user space register access. Further, these approaches need to apply various optimizations to reduce the MPK invocations for a practical performance overhead. Both problems hinder their adoption, especially in JIT engines (just-in-time compilers). Donky [165] does not rely on MPK, and hence, avoids the user space access problems but also has more PKU interactions than, e.g., ERIM. IMIX avoids all these problems 1) by preventing permission changes to safe regions after they have been set up and 2) by allowing fast, frequent domain switching.

Hardware-based Capability Systems. Capability systems [202, 58, 173] extend the hardware to enable complex memory-access policies. This is commonly done by augmenting the memory resource (granularity can vary among the approaches) with dedicated tag bits, similar to the domain concept in MPK. A particularly flexible approach is CHERI [202], an ISA extension that adds a new hardware data type to support secure pointer manipulation while providing fine-grained memory protection and access control. The CHERI system relies on software policies to express the allowed memory resources (bounds and permissions) referenced by each pointer. To fully realize the benefits of CHERI, extensive software support is required at various levels, including the operating system, compiler, language runtime, and applications, as well as the underlying microarchitecture for check enforcement.

Similarly, HDFI [173] augments the MMU with an additional tag table to realize fine-grained memory access policies. However, this approach requires reading the tag table for each memory read/write, and hence, needs additional hardware units (e.g., caches) to reduce the performance impact.

Finally, PUMP [58] proposes another tag-based capability system but extends all data units to fit the tag directly in the unit itself, avoiding the additional tag table read. While PUMP is a promising approach for enforcing security policies, it also introduces a significant hardware area overhead of around 110%.

However, these systems cannot easily support safe regions for mitigations, as their complex access control prevents frequent domain switches [135], and, unlike IMIX, these approaches are not fail-safe, i.e., uninstrumented code can bypass these policies [218].

PHMon [54] is another approach that uses a programmable hardware monitor to enforce rule-based security policies at run time. PHMon's rules cover the entire predefined architectural state of the processor, including the current instruction, used data, and program counter. PHMon is directly connected to the processor pipeline and receives traces of the processor's architectural state, which are compared against the defined rules. If a violation is detected, an interrupt is triggered.

Yet, PHMon is a reactive approach as it relies on the processor trace and can only detect an attack after it has happened. In practice, this can be enough to escape the actual monitored application and invoke unintended but benign-looking functionality. IMIX already prevents the actual data reads, making it a proactive solution with minimal hardware changes.

3

Flexible Enclaves for Application-driven Security

Today, the software stack of a computer system typically comprises an operating system (OS), libraries, runtimes, and a plethora of different services and applications from numerous vendors and open-source projects. A single vulnerability in one of these software components could compromise highly sensitive applications like payment wallets, digital rights management services, or even the whole system. As strong isolation between all software components is not feasible due to their interaction, at least the sensitive applications need to be protected from the rest of the system. This is a traditional application for Trusted Execution Environments (TEEs), sometimes also referred to as enclave architectures, where a security-sensitive application is protected from the remaining software stack within a so-called enclave. While isolating a service with a TEE appears to be a trivial solution for this problem, the diversity of today's software components creates a tremendous challenge. Each sensitive application has unique requirements on features and resources, e.g., communicating with a peripheral securely (like the fingerprint sensor in a phone), multi-process computation with inter-process communication, or integrity- and confidentiality-protected file accesses. Enclaves can leverage the regular operating system to fulfill these requirements, but this directly contradicts the idea of not trusting the other software components on the system.

Fortunately, these unique requirements of sensitive applications have already been studied extensively, and numerous enclave architectures have been proposed to handle a particular type of service and its usual requirements—based on what the enclave architecture designer deemed relevant.

In practice, three major types of enclaves can be used to protect a sensitive application, but usually, only a single enclave type is provided by the TEE. User space enclaves isolate individual user processes, which keeps the enclave's attack surface small by omitting unneeded software components like an operating system. However, this also prevents the application from performing more complex tasks, e.g., accessing peripherals, as there is simply no design feature to allow drivers to run within an enclave. Another type of enclave architecture isolates not only a user process but an additional runtime or kernel (which is then called a kernel-space enclave). While this enables a sensitive application to interact with devices like machine-learning accelerators directly, having a kernel inside an enclave creates a lot of security and management overhead for simple applications. Also, other OS components are now part of the attack surface and need to be updated regularly.

Finally, virtual machine (VM) enclaves, the enclave type found in current-generation commercial TEEs like AMD SEV, isolate whole virtual machines. However, these isolated VMs then depend on services the untrusted hypervisor provides. Typically, these TEEs cannot establish direct secure communication between an enclave and a peripheral.

While there are workarounds for each enclave architecture to emulate one of the other enclave types to offer more enclave types on a single platform, these approaches retrofit tremendous additional software (security) features [10, 170, 187]. This increases the complexity and size of the sensitive application and also forces the developer to be aware of these security features and the associated threat model. For instance, user space enclaves can access the filesystems when leveraging workarounds like Scone [10]. However, then, they rely on OS services, making them susceptible to side-channel/controlled-channel attacks from the OS [145]. So as in the original case of the three basic enclave types, this leaves the burden on the sensitive application developer—the sensitive application has to be adapted to the TEE, not the TEE to the requirements of the sensitive application.

To address this problem holistically, an enclave architecture needs to address several challenges, namely, the approach...

1. must be flexible, i.e., offer enclave types that meet the sensitive application's requirements,

2. must offer the secure use of peripheral devices to support modern use cases like machine learning,

3. has to protect the sensitive application against sophisticated attacks,

4. while keeping the enclave runtime minimal.

3.1 Our Contributions

This thesis addresses these requirements on practical enclave architectures with the following publication, which can be found in Appendix B.

[13] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, A. Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021. CORE Rank A*. Appendix B.

CURE is a novel TEE architecture for high-performance RISC-V systems that is the first to provide multiple types of enclaves. CURE offers strong and flexible isolation that adapts to the sensitive application, not vice versa. Further, CURE enables fine-grained resource management to meet enclave resource requirements without increasing the attack surface unnecessarily. Finally, CURE protects against cache side channels and controlled-channel attacks.

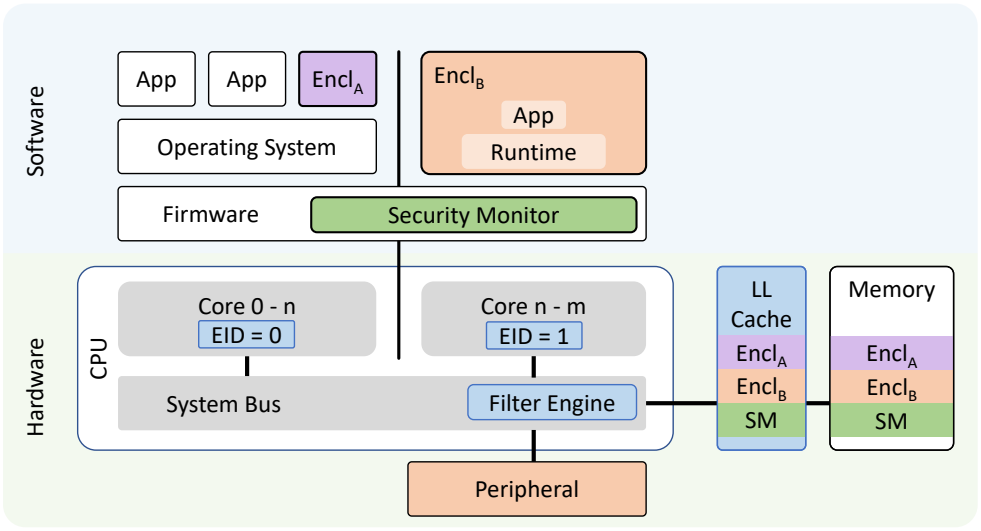


Figure 3: High-level overview of CURE. New or modified hardware components are shown in blue, the software TCB is marked green.

CURE. The key idea of CURE, as depicted in Figure 3, is to control access directly on the memory subsystem's central part: the system bus. This is the central point where memory transactions from the CPU or peripherals are routed toward the DRAM, other MMIO-mapped peripherals, or DMA devices. CURE augments the system bus by adding a Filter Engine, which comprises new access-control mechanisms for the system bus at its arbiters and decoders (the ports to the respective child components). These access-control mechanisms maintain permission tables indexed by the address range and the current enclave ID. Every memory transaction is then augmented with an enclave ID such that the Filter Engine can look up the individual permissions per transaction. This enclave ID originates from a per-core CPU register indicating the current enclave execution context. In addition, CURE introduces a new privileged software component, the Security Monitor, to configure the enclave ID and manage the Filter Engine's permission tables.

This unique hardware design enables CURE to support flexible but strongly isolated enclaves. CURE can provide different types of enclaves based on the requirements of the sensitive application: user space, kernel space, or sub-space enclaves. User space enclaves are ideal for smaller workloads that do not need peripherals or rich standard

libraries to work, thus profiting from a small TCB. CURE also effectively mitigates page- or interrupt-based controlled-channel attacks. While the OS still provides services to the user-space enclave, CURE prevents controlled-channel attacks 1) by allowing the enclave to register its own interrupt handler to detect attacks, and 2) by moving the corresponding page tables inside enclave memory. For more complex workloads, kernel space enclaves can be used. Kernel space enclaves comprise the sensitive application itself and a runtime, e.g., a kernel. This allows kernel space enclaves to contain drivers that can be used with peripherals. As CURE's Filter Engine also manages permissions for peripherals and Direct Memory Access (DMA) regions, CURE can enable direct enclave-to-peripheral bindings without requiring any changes in the peripheral. This allows kernel space enclaves to support modern workloads that, e.g., outsource machine learning to dedicated hardware accelerators. Finally, CURE supports sub-space enclaves, which isolate only a part of a sensitive application at the same privilege level. This enclave can be used to create lightweight digital rights management solutions right in the sensitive application. For CURE, a sub-space enclave is used to reduce the TEE by separating the Security Monitor from the firmware.

Implementation. We implemented a prototype of the CURE hardware extension on RISC-V using Rocket Chip, a state-of-the-art open-source RISC-V System-on-Chip (Soc) design. We extended the TileLink A and C channels to include the enclave ID signal, added a CSR register in the core, and added the Filter Engine in the system bus, where we memory-mapped the permission tables to make them configurable from software. Finally, we implemented a way-based partitioned last-level cache to prevent cache side-channel attacks.

We further implemented CURE's software part by 1) developing a Security Monitor to control the hardware, 2) adding a kernel module to the Linux kernel to handle enclave creation requests, and 3) creating a runtime for kernel space enclaves based on Linux.

The hardware overhead was evaluated by synthesizing our CURE hardware model on a Virtex UltraScale FPGA and comparing CURE to the baseline Rocket Chip. Thus, the hardware overhead is represented in lookup tables (LUTs) and registers. The TileLink extension adds 0.4% more LUTs and registers, while the access-control mechanisms add 8.6% LUTs and 3.8% registers for the main memory, 0.4% LUTs and registers for an MMIO-based peripheral, and 0.2% LUTs and 0.3% registers for a DMA device. The partitioning of the last-level cache adds another overhead of 1.7% LUTs and 1.8% registers. In summary, the changes needed to implement the CURE hardware are minimal. We further used Verilator, a cycle-accurate Verilog simulator, to ensure that our Filter Engine is fully combinational logic, i.e., operates in the same cycle.

Then, we evaluated the software changes (the security monitor and additional kernel functionality to interact with the security monitor) with microbenchmarks and macrobenchmarks based on the RISC-V ISA simulator SPIKE, and QEMU, which we both

adjusted for the additional cache and TCB flushes that CURE imposes for security. These simulators reach higher execution speeds than RTL (register-transfer level) simulators that simulate the whole synthesized circuit, while not requiring the extensive engineering work to implement, e.g., the frontend CPU to FPGA chip communication, for evaluation directly on an FPGA. The microbenchmarks showed that user space enclaves are significantly faster to set up than kernel space enclaves, even though dynamic memory allocation is slower since the Security Monitor needs to verify the user-space enclave's page tables for every new entry. In the macrobenchmarks, user space enclaves impose a geometric mean overhead of 19.70%, while kernel space enclaves only induce 15.33% overhead (compared to a normal user-space process). The macrobenchmarks run noticeably longer (as they perform more complex tasks), hence, kernel space enclaves benefit from faster dynamic memory allocations to make up for the costly enclave initialization. We further tested kernel space enclaves with the `stress-ng` benchmark, showing that multi-core kernel space enclaves scale almost the same as native multi-processing (geometric mean overhead of 0.9% with two cores). In conclusion, our performance evaluation shows that CURE is highly practical in terms of software performance overhead and hardware area overhead.

3.2 Related Work

In the following, we summarize the related work in the field of Trusted Execution Environments (TEEs). Further, given the recently introduced commercial TEEs and academic works that followed, we put our work into perspective.

Commercial TEEs and Extensions

Nowadays, almost all major processor designers/manufacturers offer processors with TEEs. Arm TrustZone [9, 210] was among the first modern TEE architectures. TrustZone is a vital part of Arm processors, which are nowadays used in mobile devices, laptops, servers, and gaming consoles. TrustZone protects against privileged software attackers from the untrusted operating system or hypervisor, but neglects physical adversaries. The core idea of TrustZone is to split the system into two worlds: a normal world and a secure world. The normal world runs the untrusted OS, such as Linux, and all non-sensitive apps, while the secure world runs a Trusted OS (TOS), which manages resources and provides services to so-called Trusted Apps (TAs). These TAs contain sensitive functions which can be invoked from the normal world. The TOS separates the TAs' user-space processes with basic inter-process isolation. Hence, the whole secure world is actually only a single enclave, as a malicious TA could provoke a privilege escalation and corrupt other TAs [63]. This endangers other TAs, most of which are provided by the vendor, such as Google's Widevine, which is why the deployment of TZ services is very restricted nowadays. Apart from TrustZone, one of the most widely deployed solutions is Intel

SGX [94, 48]. SGX offers user space enclaves based on a traditional process model, where the enclave is treated as a child process of the host application. SGX protects these enclaves against a privileged software attacker, which can reside in user space, kernel space, or hypervisor level, and even some hardware attackers, e.g., that try to snoop or tamper with memory transactions on the bus. For this, SGX introduces additional CPU microcode, small hardware changes at the Page Table Walker, and a Memory Encryption Engine (MEE) to protect enclaves from DMA attacks. The MEE encrypts the enclaves' memory and ensures its integrity, so encrypted pages cannot be rolled back or switched. SGX explicitly excludes cache side-channel attacks from its adversary model, which still can have dramatic consequences for approaches that put highly sensitive data in SGX enclaves.

AMD Secure Encrypted Virtualization (SEV) [102, 103, 6, 7] is another TEE architecture for AMD platforms. SEV leverages the AMD Secure Memory Encryption (SME) technology to isolate VM enclaves from each other and the untrusted hypervisor. SEV uses a per-VM key managed by the Platform Security Processor (a dedicated co-processor), which is also responsible for attestation. There are already multiple versions of SEV, as the initial SEV design did not encrypt the processor register state, which was then the target of controlled-channel attacks. The second version, SEV-ES (Encrypted State) [103], covers these, but there still was no integrity protection for VM enclaves' pages. As a result, various controlled-channel attacks on SEV were presented. Finally, to guarantee the integrity of pages, SEV-SNP (Secure Nested Paging) [7] introduces an additional address translation to determine page ownership, building on the newly introduced Reverse Map Table. This mechanism allows only the page owner to modify a page.

Based on the high-level ideas of AMD SEV, the newly introduced Intel TDX [93] uses per-VM encryption to isolate enclaves. In contrast to SEV, TDX uses an encryption scheme that also ensures the integrity of the enclave memory.

Finally, Arm Confidential Compute Architecture (CCA) [128] was proposed, following AMD and Intel's trend of VM enclaves. CCA is a multi-enclave architecture that introduces so-called realms. Each realm is its own VM enclave, completely isolated from other enclaves and interfaced through a privileged realm manager. This realm manager, and the enclaves, are also protected from the hypervisor. The isolation is enforced by a newly added translation level in the MMU. CCA further introduces memory encryption to protect enclaves against physical attacks.

Academic Approaches

In this section, we will give an overview of academic approaches on trusted execution. A broad research direction is realizing lightweight security architectures on smaller embedded systems, which provide security services such as attestation with minimal or no hardware changes. An example of the early works in this area is Flicker [131], which

combines a Trusted Platform Module (TPM) with a dynamic root of trust (in this case, AMD Secure Virtual Machine) to isolate small pieces of code.

SMART [64] establishes a dynamic root of trust functionality by storing the attestation code in read-only memory (ROM) and a securely stored key that can only be accessed if the program counter is pointing to the ROM region. SPM [179] and its follow-up works, such as Sancus [140, 141] extend the Memory Protection Unit with additional permission checks to reduce the TCB to the hardware itself. TrustLite [106] introduces an Execution-Aware Memory Protection Unit (EA-MPU) to define memory access policies based on the position of the program counter, allowing TrustLite to generalize the concept of SMART for independent regions. TyTAN [25] extends this approach with real-time support, secure boot, and secure storage.

However, these approaches are limited in their ability to support modern computing needs, such as multi-core setups, complex resource management, and secure peripheral access. Therefore, there is also research on approaches for more powerful computer systems that can handle more complex code.

Based on widely available virtualization extensions, a large body of work uses virtualization as a primitive for shielding sensitive applications [208, 39, 117, 159, 130, 84]. However, in multi-enclave scenarios, the hypervisor must provide typical virtualization features like para-virtualized device access (to allow all enclaves to access peripherals) and serve as the security monitor. This concentrates a lot of responsibilities in a single component, leading to a larger TCB [39, 208, 84]. Further, virtualization also slows down the untrusted OS.

Another popular research direction is extending commercial TEEs for more versatility and security. For TrustZone, most of these extensions aim to provide new types of enclaves [77, 29]. In particular, TrustICE [180] proposes a TrustZone-based concept for temporal isolation in the normal world. TrustICE suspends the regular OS for every sensitive application execution and invokes the TrustZone secure world to reset the normal world to a trusted state. After execution of the sensitive code, TrustICE cleans up the execution environment and resumes regular OS execution. While TrustIce uses only temporal isolation, SANCTUARY [29] adds spatial isolation to create deprivileged enclaves in the normal world, and hence, does not have to suspend the OS for enclave execution. SANCTUARY creates strongly isolated enclaves by using the TrustZone Address-Space Controller (TZASC) to assign memory regions based on IDs. While this feature is commonly used to DRM-protect media being transferred between CPU and GPU, SANCTUARY assigns IDs to individual CPU cores for fine-grained memory isolation. Other works harden TrustZone against new attack vectors, e.g., by encrypting the secure world memory area to prevent physical attacks [214], or by moving the secure world to a dedicated chip [215]. However, as these approaches cannot change the underlying security primitive used to construct the TEE, efficient side-channel resilience,

and binding a peripheral directly to an enclave in multi-enclave settings remained an open challenge [13].

In order to change the underlying primitive, more control over the hardware is necessary. As such, RISC-V emerged as a promising foundation for academic TEE research. One of the first TEEs on RISC-V was Sanctum [49], which extends the RISC-V architecture to create user space enclaves similar to Intel SGX. Sanctum builds on dedicated page tables per enclave and minor changes to the Page Table Walker to protect these page tables. A modified page table walker (PTW) prevents address translation for virtual addresses mapping to physical addresses used by other enclaves. Sanctum does not encrypt enclave data but can restrict DMA accesses.

Keystone [114] instead provides kernel space enclaves using the RISC-V Physical Memory Protection (PMP) to create isolated memory regions per enclave. Keystone offers a small runtime for driver access directly from enclaves, yet, does not support a direct enclave-to-peripheral binding. Further, Keystone does not isolate its security monitor from the firmware, creating a larger attack surface.

Another work, SERVAS [176], leverages authenticated encryption based on MEMSEC [201] to isolate memory. SERVAS can also protect against physical rollback/replacement attacks by ensuring authentication. In contrast to CURE, SERVAS only supports user space enclaves, does not offer enclave-to-peripheral bindings, and cannot protect against interrupt-based controlled-channel attacks.

Finally, Penglai [66] is one of the most recent RISC-V TEE architectures. Penglai combines page-table-based isolation with memory encryption (with a scheme that also provides integrity) and cache partitioning to provide user space enclaves. In comparison to other RISC-V TEEs like KeyStone, Penglai uses a safeguarded page table to isolate the memory: 1) the page table is moved to a dedicated memory section, 2) the PMP is configured to protect this table, and all memory accesses to this region are trapped by the security monitor, 3) the page table walker (PTW) is extended to only allow page table walks within the protected memory area. This high-level concept is quite similar to Sanctum's memory isolation. In contrast to CURE, Penglai only supports user space enclaves and has no support for enclave-to-peripheral binding.

4

Enclaves as Security Primitives in Protocols

In the preceding chapters, the focus was on attack-surface reduction techniques that utilize isolation to protect highly-sensitive software. However, these techniques can also serve as a fast and efficient security primitive within complex protocols. For instance, homomorphic encryption is a powerful cryptographic solution, but its computational cost is high. Trusted Execution Environments (TEEs) can significantly accelerate homomorphic encryption in protocols [61, 197, 138].

Similarly, TEEs can speed up multi-party computation (MPC) [194, 142, 204], another cryptographic protocol known for its computational complexity.

Furthermore, TEEs offer the ability to perform remote attestation, a cryptographic method for verifying the correct setup of a sensitive application. The combination of isolated execution and remote attestation allows TEEs to further simplify protocols and reduce their complexity.

In this chapter, we will explore two additional areas that stand to benefit from the implementation of TEEs: an off-chain smart contract execution facility for blockchains and a secure offline machine-learning inference protocol. These areas highlight the versatility and usefulness of TEEs as a security primitive in complex protocols.

4.1 Efficient Off-Chain Smart Contracts

Since Bitcoin was introduced in 2009, the blockchain ecosystem has evolved continuously. Nowadays, a plethora of different blockchain-based services exist, ranging from payment systems, over ownership certificates in the form of non-fungible tokens (NFTs), to permissioned blockchains used in corporate settings. One crucial cornerstone in this ecosystem is Ethereum, a blockchain that allows the execution of so-called *smart contracts*, computer programs that can trigger actions, e.g. payment transactions. Ethereum aimed to become a decentralized world computer [151], however, after a decade of improvements to the blockchain ecosystem, typical smart contracts on Ethereum are still surprisingly simple. In fact, most deployed contracts are simple token managers [146].
The underlying reason is that Ethereum and many other blockchains have scaling issues

due to 1) contract execution being replicated on multiple miners, 2) high execution fees, and 3) slow confirmation of transactions.

Hence, off-chain smart contract execution emerged as a key research direction to address this problem. These approaches move smart contract execution off-chain to minimize costly interactions with the blockchain.

The most prominent areas in this research direction are second-layer solutions. One of them is state channels [132, 62], which allow users to lock funds, communicate off-chain for money transfers, and split the locked funds based on the communication outcome. This is very efficient in the optimistic case but requires costly on-chain conflict resolution in case of a dispute. Another approach is Plasma [155], which creates a separate blockchain anchored to the main chain to speed up transaction processing. However, Plasma does support smart contracts. Similarly, Rollups [184] also process transactions off-chain but publish their results on-chain. A significant disadvantage of rollups is that centralized rollup operators can influence transaction ordering. Others use a quorum of execution agents to approve the results of contract executions, yet, they either do not support private state or require on-chain communication for every contract invocation [206]. Finally, some approaches leverage TEEs to secure the off-chain computation [41, 52]. These approaches suffer from a classic trusted computing problem: isolated enclaves are perfectly capable of protecting a program, but an enclave might just never be scheduled (again) by the regular OS to actually run it. Hence, smart contracts are not guaranteed to finish execution; hence, existing enclave-based approaches cannot guarantee liveness.

TEEs are, due to their native computation speeds, a compelling building block for off-chain computation. Yet, an off-chain approach based on TEEs needs to address several challenges, namely, the approach must...

1. not require parties to lock a large collateral,
2. avoid frequent blockchain interactions,
3. offer entering and leaving the instance,
4. tolerate flexible contract lifetimes,
5. while keeping the contract state private.

Our Contributions

This thesis addresses these requirements on TEE-based off-chain execution with the following publication, which can be found in Appendix C.

[68] Tommaso Frassetto, Patrick Jauernig, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi. POSE: Practical Off-chain Smart Contract Execution. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 24-27, 2023*, 2023. CORE Rank A*. Appendix C.

POSE is a novel off-chain smart contract execution protocol with strong liveness guarantees and private state without relying on collateral. Further, POSE offers native computation speeds for smart contracts, bringing the ever-envisioned world computer closer by enabling modern use cases such as machine learning in smart contracts.

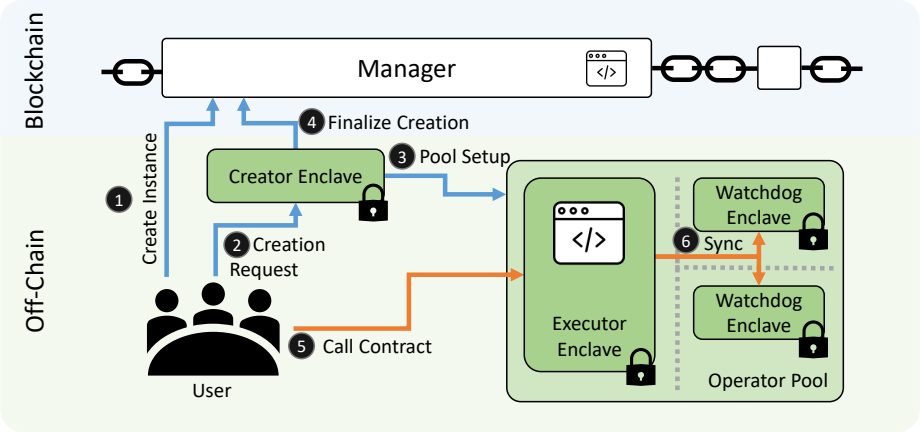


Figure 4: High-level overview of POSE

POSE. POSE leverages a TEE-based execution pool, consisting of an executor enclave and multiple watchdog enclaves, to take over the execution on the failure of the executor and ensure execution of the smart contract while maintaining private state. As shown in Figure 4, to set up a POSE contract, ❶ a set of enclaves is first registered with the on-chain POSE manager contract. An enclave provider signs a confirmed block header to prove sufficient synchronization with the blockchain. Then, ❷ a user can create a contract instance with the manager's help. For such a request, the manager chooses an enclave out of the set, which is the creator enclave. Next, ❸ the creator enclave sets up an execution pool for the contract, consisting of randomly selected enclaves from the set: one executor enclave to execute the contract and multiple watchdog enclaves that can replace the executor to ensure availability. POSE's manager contract assigns a pool of enclaves to a smart contract instance. Now, ❹ the creator enclave submits finalized contract information to the manager (identities of the selected pool enclaves and their roles) such that ❺ the user can send inputs to the executor. Finally, ❻ the executor performs the user's contract call and distributes the resulting state to the watchdogs, which acknowledge the update (without re-executing the contract call on their side). Unresponsive executor operators can be challenged over the blockchain. If the operator fails to respond, the operator is dropped from the pool and a watchdog takes over.

Implementation. We implemented a prototype of POSE consisting of an Ethereum smart contract for the manager, Arm TrustZone [9] to realize the smart contract execution facility, and Lua as the smart contract programming language. The manager is implemented as an Ethereum smart contract written in Solidity. The manager does bookkeeping of the pools and contracts, while also offering functions to register an enclave, create a new contract instance, deposit or withdraw funds, and challenge an enclave on-chain. The enclaves are implemented using Arm TrustZone. Arm TrustZone is a Trusted Execution Environment that supports two execution modes: in the so-called normal world, the regular OS with its applications is running, while the Trusted OS and its Trusted Applications reside in the so-called secure world, which is only used for security-sensitive applications (e.g., the fingerprint reader service on a smartphone). Both worlds can communicate over shared memory. As such, we implemented POSE as a Trusted Application in secure world using OP-TEE [123] as the Trusted OS. We chose Lua as a smart contract programming language as the Lua interpreter is relatively simple and only has standard library dependencies. We ported the interpreter to TrustZone by stripping out unsupported operations, e.g., filesystem accesses. During the setup phase of the enclave, the Lua interpreter is initialized with the contract. Then, OP-TEE is used to generate an attestation report as a confirmation to the creator enclave. At run time, the TA exposes a `get_input()` function to Lua—when called, the TA creates a snapshot of the Lua state, returns to normal world to receive the input via the network, switches back to secure world, restores the Lua state, and injects the new input. To interact with the manager contract, the TA utilizes an Ethereum wallet designed for embedded devices [8].

4.2 Private and Secure Offline Machine Learning

Another area that is benefiting from TEEs is machine learning. Every aspect of our mobile device experience is slowly enhanced by machine learning (ML): the keyboard predicts the next words, the app launcher suggests suitable apps based on previous user behavior, or powerful voice assistants that aid the user. All these ML-based services are increasingly turning to core technologies, e.g., Google's camera improvements in phones can largely be attributed to advances in AI-based image enhancers [22].

Still, many of these services require frequent cloud interaction. For instance, voice assistants often parse the query locally but look up the actual result online to provide a result. Another example is the keyboard sending feedback on the word prediction feature to the cloud to improve the general prediction model [209]. This is a tremendous privacy risk for users, as it is not transparent which data gets transmitted at which time.

Counterintuitively, offering AI-based offline services also poses risks for the vendors. ML algorithms are nowadays running on the device itself, using dedicated hardware-based

accelerators. This guarantees fast response times for the user and reduces the load for the cloud infrastructure.

However, when the model is directly stored on the device, this also exposes the machine-learning model to model-stealing attacks. As explained previously, the ML model is nowadays often at the core of a service offering, the model is precious intellectual property to the vendor. Hence, model stealing attacks can have grave consequences.

A security mechanism is needed to protect the vendor's assets, but also to prevent user data from leaving the device without consent.

While cryptographic approaches like homomorphic encryption [147, 73, 19] have been used in previous works, they are inherently limited in performance [150] or require the model to be adjusted [160]. Further, existing approaches using Trusted Execution Environments [26, 90] 1) only either protect the vendor's asset or the user data, not both, 2) are designed for cloud environments, and 3) do not end-to-end protect the user input.

Therefore, a high-performance approach is needed to protect the machine-learning model and the user data while ideally still supporting hardware accelerators for machine learning. This is highly challenging, as the approach needs to...

1. maintain a two-way isolation between the machine-learning model and the user's data and applications

2. provide a secure binding between the machine-learning model and accelerator hardware

3. ensure that the vendor can always control access to its model, i.e., prevent access, e.g., when a license is revoked.

Our Contributions

This thesis addresses these requirements on secure and private machine learning with the following publication, which can be found in Appendix D.

[17] Sebastian P. Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. Offline Model Guard: Secure and Private ML on Mobile Devices. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 460–465. IEEE, 2020. https://ieeexplore.ieee.org/document/9116560. CORE Rank B. Appendix D.

OMG is a novel secure and private machine learning approach even in offline scenarios. OMG leverages SANCTUARY [29], a Trusted Execution Environment, to guarantee two-way isolation between model and user. Further, OMG utilizes SANCTUARY's peripheral binding to allow hardware-based machine-learning accelerators to be used for inference. Finally, OMG features a model manager that loads the vendor's model and manages access rights for the model and the input devices used to generate inputs for the model.

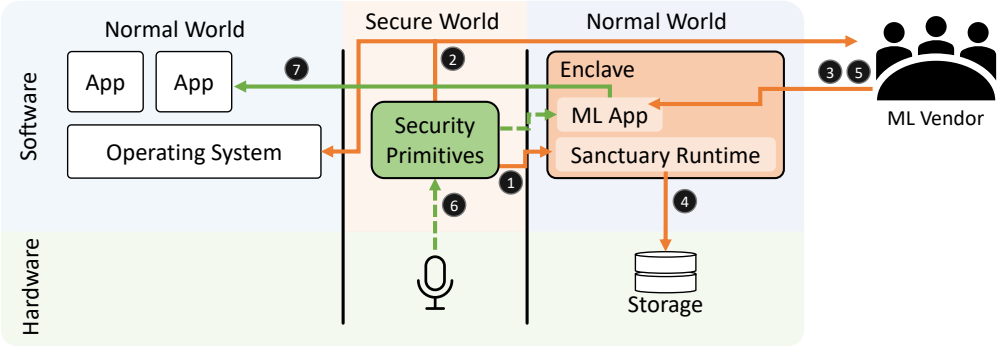


Figure 5: High-level overview of OMG

OMG. OMG is a TEE-based protocol for secure and private machine learning. As depicted in Figure 5, OMG's protocol consists of three phases: preparation, initialization, and operation. In the preparation phase, ❶ the enclave runtime is loaded and ❷ its correct setup is attested towards the user and the vendor using the TEE's remote attestation functionality. Only then, ❸ the vendor deploys the encrypted ML model to the enclave (which ❹ can be stored in persistent storage by the enclave to allow offline use). In the initialization phase, ❺ the vendor can (based on previously exchanged keys) provide the decryption key to the enclave. This key changes with every major release of the ML model to allow for digital rights management. Finally, in the operation phase, ❻ the user can provide input data, e.g., voice, to the enclave. As peripherals can be bound to an enclave with SANCTUARY, the user can securely send microphone input over the Security Primitives to the enclave. Subsequently, ❼ the enclave provides the output of the ML model, e.g., text, to the user.

Implementation. We implemented a prototype of OMG based on TensorFlow Lite for Microcontrollers [2] on the Hikey 960 development board, an 8-core ARMv8 System on Chip with 3GB of RAM, and TrustZone support. We set up SANCTUARY accordingly to provide normal-world enclaves with peripheral bindings. We further implemented an offline keyword recognition application to classify input audio based on pre-set trigger words. The ML model comprises a 2D convolutional layer, a rectified linear unit (ReLU) activation, and a regular mapping layer. Model training is performed beforehand based on 105,000 WAVE audio files (voice samples of 30 different commands) [199]; the resulting model is 49KB. For our performance evaluation, we measured the time needed for a

single keyword interference in a SANCTUARY enclave versus an interference outside of the enclave. Setting up the SANCTUARY enclave (until the start of the runtime) takes around 300ms (exact measurements can be found in [29]), while during inference, SANCTUARY imposes only around 2% performance overhead over native execution due to the switch to the secure world to fetch input data. Inference accuracy remains identical. Our prototype implementation and performance evaluation demonstrate that OMG can provide machine learning with strong security and privacy guarantees for the user and the machine-learning model vendor at negligible performance costs on mobile devices.

4.3 Related Work

In the following, we summarize research related to POSE and OMG, and put our work into perspective. For POSE, we discuss off-chain execution proposals leveraging MPC, VMs, second-layer approaches, or Trusted Execution Environments (TEEs). For OMG, we discuss cryptographic and TEE-based approaches for secure and private machine learning.

Off-Chain Smart Contract Execution

The research on efficient smart contract execution leverages various techniques to improve performance and security guarantees: multi-party computation (MPC), state channels (and other second-layer solutions), virtual machines, or TEEs.

While some approaches use MPC to build a quorum of smart contract executors to ensure confidentiality [110, 109, 111], MPC is known for its significant overhead in communication and performance. Further, these approaches all require collateral. Hence, we will focus on second-layer and TEE-based approaches here.

Second-layer Approaches. Second-layer approaches [132, 155, 62, 184], as mentioned before, enable parties to lock funds on the blockchain such that the funds can be redistributed off-chain. When the off-chain communication is finalized, the locked funds are distributed accordingly. This reduces the number of transactions needed but is costly in case of a dispute (which has to be solved on-chain). Further, second-layer approaches primarily focus on optimizing monetary transactions, hence, most second-layer protocols do not support smart contracts.

VM-based approaches treat a smart contract as a virtual machine that is executed by a quorum. Arbitrum [101] requires that all managers agree on the outcome—the result can then be signed and posted on-chain. ACE [206] and Bitcontracts [205] improve on this by reducing the needed confirmation to a subset of the quorum; however, they all do not

support private state (quorum members have full access to the VM's data) and require frequent blockchain interactions.

TEE-based Approaches. Two promising methods for leveraging Trusted Execution Environments (TEEs) to enhance smart contract performance are 1) improving the speed of the blockchain's underlying consensus mechanism with the use of TEEs [126, 78, 195], and 2) offloading the actual execution of smart contracts to TEEs, as done in POSE. In contrast to consensus improvements, TEE-based offloading further allows for compatibility with existing, legacy blockchains, while enriching smart contracts with native computation capabilities. Here, we focus on offloading techniques, as they are more related to POSE.

Ekiden (and its successor, the Oasis Network [67]) leverages multiple TEE-based compute nodes that execute the smart contract and manage cryptographic keys. However, Ekiden requires the (encrypted) state updates to be sent to the blockchain after every function call. This makes Ekiden off-chain executions very costly. Avalon [91], an industry proposal similar to Ekiden features the same shortcomings. FastKitten [52] tackles off-chain execution using an incentive-driven protocol against rational adversaries, i.e., financially motivated. In FastKitten, a single operator uses a TEE to execute the smart contract, incentivized by a deposit to execute the complete smart contract. However, this deposit must be as high as all participant's deposits combined. Further, only a fixed set of participants and a limited contract lifespan is possible in Fastkitten.

POSE overcomes these limitations. Its strong liveness guarantees prevent frequent blockchain interactions while also avoiding any collateral at all in the general case. In addition, POSE supports long-lived contracts with dynamic sets of participants and protects against a stronger adversary than FastKitten.

Secure and Private Machine Learning

A machine-learning model can be protected by various means, either by keeping it on a remote server (and ensuring the privacy of inputs) or by protecting the model on-device to classify inputs locally without leaking the model. Both aspects have been the subject of extensive research.

Cryptographic Approaches. Cryptographic approaches typically leverage homomorphic encryption, two-party computation, or (secure-)multi-party computation. Homomorphic encryption (HE) is a relatively recent addition to the numerous cryptographic primitives. HE can apply an operation directly to encrypted data without leaking information from the data. While this powerful primitive has been leveraged in several approaches [147, 73, 19], HE is a highly computationally intense primitive, preventing

these approaches from scaling to bigger models [150]. Multi-party computation (MPC) is another existing approach to secure machine-learning models [162, 14]. A subset of MPC, two-party computation, has proven to be more efficient and has been more widely used [136, 125, 161, 100, 160]. However, these approaches need the model to be adjusted based on the security primitive used, and hence, require tedious manual tweaks by experts.

In contrast, OMG supports unchanged machine-learning models without any additional requirements on the model such as its size.

Enclave-based Approaches. In contrast to cryptographic approaches, TEE-based approaches can maintain the performance characteristics of the machine-learning model while providing additional security guarantees [185]. One of the first TEE-based approaches was proposed by Ohrimenko et al. [143] based on Intel SGX. Ohrimenko et al. aggregate encrypted data from multiple data providers on a single server to train a machine-learning model. To prevent information leakage during the training—the only time the data is unencrypted—they propose to perform the aggregation and training in an Intel SGX enclave. As SGX enclaves have been shown to be vulnerable against side channels based on data-dependent access patterns [27], the authors present data-oblivious variants of prominent ML algorithms. Similarly, Myelin [90] creates a privacy-preserving model graph that is then trained in a server-side SGX enclave. Privado [76] also protects against side channels based on memory access patterns. Another work, Chiron [88], extends the underlying concept of these works by allowing the machine-learning-as-a-service provider to freely select, configure, and train ML models. SecureTF shares the same goal but provides a more general ML framework based on Scone [10], while Occlumency [115] aims to speed up interference in SGX enclaves. Another approach, Pr$\epsilon\epsilon$ch [5], combines the security of SGX enclaves with differential privacy—at the cost of lower accuracy. VoiceGuard [26] focuses on privacy-friendly inference using an SGX enclave. User data from smart home devices is s sent to a service provider where ML models process the data. By keeping the model in an enclave at all times, model stealing attacks are prevented efficiently. At the same time, VoiceGuard ensures that the user data is not used for purposes other than inference. MLCapsule [80] has the same goal, but brings the service provider functionality directly on the user's device, and hence, supports offline scenarios. Other works used Arm TrustZone as a TEE [127, 181], however, this increases the attack surface of the system TCB significantly, especially as drivers for hardware accelerators would need to reside in the secure world.

Naturally, forcing ML models to run on CPUs is quite restricting, as modern models are huge and performance-hungry. Therefore, another research direction is to leverage GPUs securely for ML models. Slalom [185] and eNNclave [163] both split the ML computations into sensitive and non-sensitive parts, where only the non-sensitive parts are outsourced to the GPU. Telekine [87] uses a GPU-based TEE (cf. Graviton [192]) and provides data-

oblivious encrypted data streams for communicating with the GPU. In Visor [154], only the inference is moved to the GPU TEE, while user input processing is done in SGX.

All these approaches either rely on SGX [185, 163], and therefore, cannot offer secure peripheral access (e.g., to the microphone), increase the TCB significantly or require a GPU TEE [87, 154], which makes them impractical for mobile devices. In comparison, OMG leverages SANCTUARY, which does not require any hardware changes and can be implemented on current Arm architectures. Further, OMG can protect ML applications from controlled- and side-channel attacks without modifying the ML algorithms. Finally, OMG can directly and securely receive user input from a sensor thanks to SANCTUARY's TrustZone support.

5

Attack Surface Analysis with Fuzzing

As outlined in the introduction, memory-corruption vulnerabilities remain a major threat to applications. In the previous chapters, we introduced enclaves as a method to protect sensitive applications from such attacks. While enclaves reduce the attack surface of a sensitive application, the sensitive application can still contain vulnerabilities itself [43]. Such vulnerabilities remain reachable by adversaries outside of the enclave, as the sensitive application usually exposes an interface to interact with other services. Further, depending on the enclave architecture, privilege escalation attacks can endanger the whole system [51, 71, 177, 156]. While mitigations within enclaves can significantly reduce the probability of a successful attack, they also come with a significant performance overhead [27].

Therefore, the primary objective in ensuring the security of a sensitive application should continue to be the avoidance of deploying software with vulnerabilities. A promising approach to do this at scale is fuzzing. Fuzzing is a dynamic analysis approach to find bugs by repeatedly executing the target (the software or hardware under test) with randomly generated inputs (testcases). These testcases originate from a so-called seed corpus, an initial set of inputs (seeds), e.g., based on previously existing unit tests. Fuzzing became an essential cornerstone of modern software testing [59] and is a broad research area. Fuzzing research has focused on making new targets fuzzable, improving integration with existing tooling, or increasing the efficiency of the fuzzing process. Especially important is the latter, as advances in efficiency reduce the time to find the same bug across targets. While technical improvements like better coverage tracing took the stage in early fuzzing research, there is a strong focus on algorithmic advances.

A recently emerging researcg area is mutation scheduling. In mutation-based fuzzers, biology-inspired mutation operators are applied to mutate testcases. These mutation operators, e.g., delete or add a random byte in the testcase, or negate a bit. There are already various approaches that determine the time that should be spent on a testcase [171, 211], as well as where to apply a mutation for a given testcase [119, 72]. However, the problem of choosing the adequate mutation operator over time, i.e., scheduling a mutation, is still part of active research.

MOPT [129], the most promising existing work, proposed to use a variant of Particle Swarm Optimization (PSO) to optimize the mutation operator probability distribution.

However, PSO has local and global best solutions. This requires costly adaptions to reach the global optimum at any time. Further, MOPT exposes various user-facing parameters that require expert knowledge to adapt the approach to the current fuzzing target.

Hence, the problem is still highly challenging. A successful approach for mutation scheduling must...

1. determine whether the problem at hand is only target-dependent or changing over the time of fuzzing,

2. balance improving the quality of selection and execution speed, especially as the latter is vital for fuzzing,

3. integrate well with the fuzzer, i.e., an optimal representation of the solution and a balance between exploration and intensification needs to be found,

4. be agnostic to the concrete target without adding additional parameters for users. If not, this would counteract the wide applicability of algorithmic improvements.

5.1 Our Contributions

This thesis addresses the aforementioned requirements on optimized mutation scheduling for fuzzing with the following publication, which can be found in Appendix E.

[97] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: Survival of the Fittest Fuzzing Mutators. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 24-27, 2023*, 2023. CORE Rank A*. Appendix E.

With DARWIN, we present a novel mutation scheduler for fuzzing based on the Evolution Strategy (ES) algorithm. DARWIN optimizes the mutation-selection probability distribution during the fuzzing process using the coverage feedback as a fitness function. In contrast to existing works, DARWIN only has a minor impact on execution speed and does not introduce new per-target parameters that require manual tuning.

DARWIN. DARWIN optimizes the mutation-selection probability distribution by leveraging the Evolution Strategy [79], a simple and efficient optimization approach. As depicted in Figure 6, in a typical fuzzing pipeline, the fuzzing loop starts with taking a testcase from the queue, a structure holding the seed corpus or derived testcases. This testcase is then mutated by randomly selecting multiple mutation operators. This mutated

testcase is then used as an input for the target while monitoring the coverage generated by this testcase. DARWIN utilizes this coverage information as a fitness function, to optimize the selection of mutation operators. Specifically, DARWIN uses Evolution Strategy to dynamically optimize the probability distribution of the mutation operators such that the selection of operators is not uniform anymore, but adapts to the target and the exploration within the target.

The ES in DARWIN uses an initial set of random parent solutions (solutions represent concrete probability distributions), which are then perturbated (mutated) to generate child solutions. The best-performing child in terms of newly found unique paths is then the new parent in the next iteration. DARWIN is transparent to the user, so no parameters have to be adjusted for a new target application, while fuzzer developers only have to adjust two parameters of ES that steer exploration and exploitation based on the set of mutations.

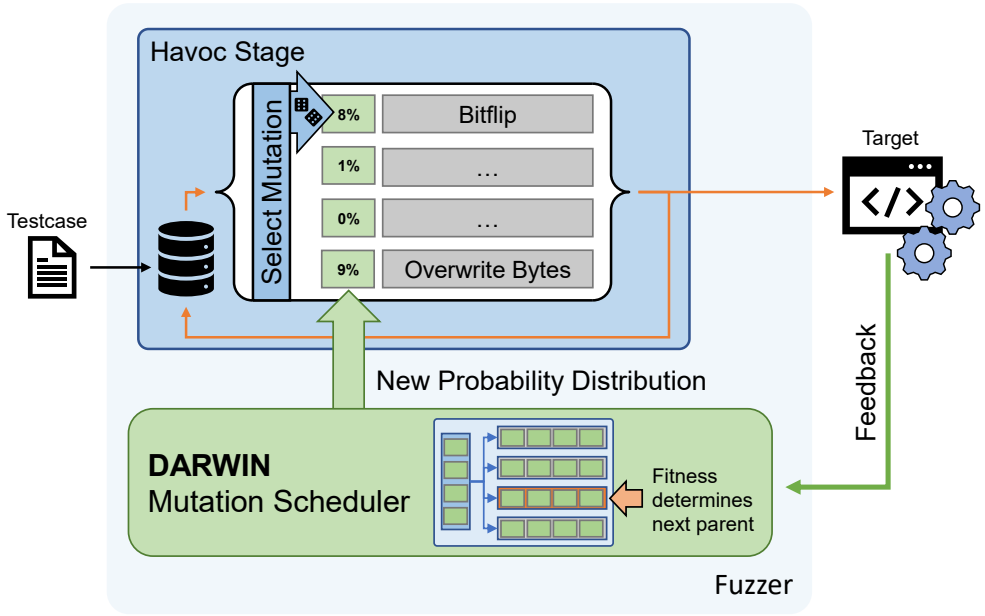


Figure 6: High-level overview of DARWIN

Implementation. We implemented a prototype of DARWIN on the popular mutation-based AFL fuzzer (version 2.54b) [74] in C. We selected AFL, as many works use it as a baseline, as it does not leverage additional optimization algorithms that might interfere with DARWIN. Our prototype adds around 300 lines of code and exposes three interface functions to the fuzzer: initialization, retrieving the next mutation operator, and reporting feedback to DARWIN. In addition, we implemented a version of AFL that can load statically set mutation selection probabilities (optimized probability vectors from regular DARWIN experiments) to evaluate whether dynamic adaption of probabilities leads to better results. The evaluation of DARWIN and its variant with a static, pre-optimized

probability distribution shows that the problem of mutation scheduling is indeed dynamic, i.e., the optimal probability distribution changes over time. Further, DARWIN outperforms baseline and related work in coverage, the time needed to achieve the same coverage, and the time needed to reach known bugs. In addition, we implemented a version based on EcoFuzz [211] to show orthogonality to seed scheduling algorithms. Finally, DARWIN uncovered 15 out of 21 bugs fastest in the MAGMA [82] benchmark, and found a new bug in GNU binutils that persisted for ten years[1].

5.2 Related Work

Fuzzing is a broad research area, ranging from technical advances like resetting the state efficiently, to algorithmic improvements within the fuzzing process. Here we restrict the overview to the latter, as this is also the focus of the work.

Mutation Strategies

There are two factors in mutation selection: when to apply which mutation and where in the testcase the mutation should mutate a byte. In the following, we will give an overview of both areas.

Mutation Scheduling. Initially, Drozd et al.[60] as well as Boettinger et al. [33] leveraged reinforcement learning to mutation scheduling, however, they failed to demonstrate coverage improvements. MOPT [129] was the first approach to claim actual improvements in coverage and bug-finding capabilities. MOPT uses Particle Swarm Optimization (PSO) to optimize the probability distribution of the mutation scheduler. PSO in its original variant is prone to converging towards local optima (so globally sub-optimal solutions), hence, the integration of the algorithm in MOPT is more complex. Consequently, MOPT requires the fine-tuning of various user-defined per-target parameters, making it harder to use in practice. In addition, the complexity of the algorithm leads to a significantly reduced execution speed, which negatively impacts achieved coverage over time. In contrast to MOPT, DARWIN's Evolution Strategy is a simpler but faster algorithm that exceeds MOPT's and the baseline's coverage as well as bug-finding capabilities without any user-facing parameters.

A parallel work by Wu et al. [203] proposes a two-layered Multi-armed Bandit (MAB) approach for mutation selection. First, a MAB is used to optimize the stacking size, i.e., how often the mutation should be applied to the testcase during the actual scheduling. Second, another MAB optimizes mutations of which group (unit- or chunk-based) should be selected uniformly. However, 1) the operator selection optimization is very coarse-

1 https://sourceware.org/bugzilla/show_bug.cgi?id=29233

grained (with DARWIN we show that fine-grained optimization leads to simultaneous high shares for operators from both groups), and 2) both MABs are not evaluated independently.

Location Optimization. Even in a setting where the ideal probability distribution of the mutation operators is known, many fuzzing iterations are needed to determine which bytes of the testcase the mutation operators should be applied to. A straightforward way to address this issue is taint tracking, i.e., choosing an interesting block in the application code and inferring which bytes need to be manipulated to drive the control flow to this block. However, due to the complexity of control flow in real-world programs, it is computationally expensive to track all related variables and input bytes [119, 72]. Instead, FairFuzz [116] uses a preliminary deterministic combination of mutations to find byte locations that lead to low-frequency paths when mutated, while Rajpal et al. [157] propose a neural network to identify interesting bytes in a testcase. Steelix [118] extracts comparisons in the program code statically to identify corresponding bytes in the testcase. In future research, these approaches could be studied in combination with DARWIN to optimize both the byte location and the set of mutators to apply.

Seed-selection Algorithms

In contrast to mutation-strategy optimizations, seed-selection algorithms aim to reduce the number of seed cases to an optimal subset with respect to a specific optimization goal (e.g., to target low-frequency paths or remove redundancy). For instance, MoonShine [148] analyzes system call traces of real-world programs to reduce redundancy and waste less fuzzing iterations on highly similar testcases. Similarly, Nichols et al. [139] train a GAN (Generative Adversarial Network), a neural network, with the initial corpus to produce higher-quality seeds. Another research direction optimizes the order in which seeds are used to fuzz. EcoFuzz [211] divides seeds into exploration and exploitation classes to ensure that newly found paths can be exploited optimally—and fully exploited paths lead to a period of explorative search. AFLFast [21] uses a Markov model to prefer seeds leading to low-frequency paths, while AFLGo [20] prioritizes seeds that are close to a target location. Another work, VUzzer [158], finds hard-to-reach paths while avoiding error-handling code by prioritzing testcases using an evolutionary algorithm, while NeuFuzz [198] uses a neural network to focus on error-prone paths. Leveraging the previously mentioned taint-tracking techniques, Angora [38] prefers inputs leading to unexplored branches. Finally, to also increase the probability of semantically correct testcases, AFLSmart [153] leverages a structural representation of seeds. Compared to DARWIN, seed-selection algorithms optimize a very early step in the fuzzing workflow. It is not clear whether later stages, especially mutation schedulers, could tend to counteract the desired optimization goal.

6

Conclusion & Outlook

In this chapter, we summarize the contributions of this dissertation to the field of isolated compartments and dynamic software analysis and give an outlook on future research directions.

6.1 Conclusion

This dissertation significantly contributes to the research field of attack surface reduction by proposing approaches ranging from data compartments within an application to safeguarding whole virtual machines. We further use these attack surface reduction techniques to construct powerful protocols in other research fields. This dissertation further contributes to the research field of attack surface analysis by algorithmically improving dynamic software analysis, specifically fuzzing, to uncover programming errors leading to vulnerabilities throughout the software stack. In the following, we summarize the contributions of this dissertation in detail.

Secure In-Process Compartments. We presented IMIX [70, Appendix A], an in-process memory isolation technique supporting high-frequency domain switches to protect modern software defenses. We implemented a prototype to protect the metadata of Code Pointer Integrity using Intel Wind River Simics, an x86 full system simulator. In contrast to previous approaches, IMIX has a minimally invasive design, offers deterministic protection of memory regions, and achieves a negligible performance overhead.

Flexible Enclaves for Application-driven Security. We presented CURE [13, Appendix B], a flexible enclave architecture for RISC-V that supports different types of enclaves on a single platform. CURE offers intra-privilege level, user-space, and kernel-space enclaves. We implemented a prototype of CURE on the open Rocket Chip and showed its practicality by evaluating hardware and performance overhead. In contrast to other enclave architectures, CURE adapts to the sensitive application, not the other way around. Further, CURE introduces a novel system bus filtering component, enabling secure bindings between enclaves and peripherals.

Enclaves as Security Primitives in Protocols. We presented two protocols, POSE and OMG, that leverage enclaves as a security primitive. POSE [68, Appendix C] is a novel off-chain protocol for smart contracts that features private state and strong liveness guarantees. We implemented a prototype of POSE for Arm TrustZone and evaluated smart contracts ranging from rock paper scissors to federated machine learning with 431,080 weights. Compared to previous works, POSE can ensure correct behavior without collateral thanks to its liveness guarantees. OMG [17, Appendix D] is an enclave-based protocol to protect machine-learning models and the user's privacy simultaneously. OMG leverages SANCTUARY [29], a normal-world enclave architecture for Arm platforms, to prevent model stealing attacks and manage access permissions to the model. We implemented an offline wake-word detection service by shielding the *TensorFlow lite for microcontrollers* framework inside a SANCTUARY enclave. Contrary to existing work in this area, OMG provides native machine-learning inference performance without changing the model.

Attack Surface Analysis with Fuzzing. We presented DARWIN [97, Appendix E], an efficient mutation scheduler for mutation-based fuzzers. DARWIN uses Evolutionary Strategy to optimize the probability distribution of the mutation operators iteratively throughout the fuzzing process. We implemented a prototype based on AFL and evaluated execution speed, efficiency characteristics, coverage, and bug-finding capabilities. Unlike previous approaches, DARWIN's optimization algorithm is significantly simpler and does not expose any user-facing parameters that hinder adoption.

6.2 Outlook

We have shown different approaches to attack surface reduction and analysis that can form the basis for next-generation, secure-by-design compute architectures. However, having an end-to-end secure computing architecture is far in the future. We currently see the first steps of porting existing software to capability architectures [75], which adapt capability systems directly in the instruction set. While previous works on run-time defenses often relied on exploiting hardware features in unintended ways, these next-generation primitives will be more flexible, allowing more holistic defenses. Building efficient high-level defense primitives out of these hardware features, potentially also an enclave architecture, will be crucial in the future. However, having these strong but flexible primitives is only part of the solution, as they do not guarantee the absence of security issues [24].

Further, these primitives, as well as the rest of the hardware, need to be a solid foundation for system security, and hence, must be secured. Attack surface reduction and analysis for hardware is still an emerging research field. Yet, there are promising developments in these research fields. For attack surface reduction, new flexible hardware elements, e.g.,

based on Intel's eASIC [95] will emerge, that will offer security services by interacting with the execution pipeline. For instance, a customizable instrumentation to implement heuristic, maybe even AI-based, memory corruption defenses. For attack surface analysis, hardware fuzzing is an emerging, but important field of research, as trust in software cannot go without trust in hardware that performs the computation. While basic hardware fuzzing is already available [188, 186, 89], future research will likely follow the path of traditional software fuzzing by incorporating static and symbolic approaches to examine more complex parts of hardware. Chen et. al [37] took a first step in this direction, but better integration with existing analysis tools for hardware is needed. For software fuzzing, future work should focus on how the different stages in the fuzzing process interact and how they can be optimized in combination.

Together, these building blocks will coin future computer systems. Yet, there are also short-term developments that will be equally important.

The introduction of modern, flexible TEEs, will increase their adoption in the cloud and embedded markets. Intel TDX and Arm CCA will inspire a variety of research focusing on evaluating the security guarantees of these architectures and developing higher-level security primitives for use in protocols. However, developers will also need tools to create secure enclave code. While approaches for Intel SGX [44, 43] and Arm TrustZone [81, 85] exist, these newer TEE generations enable developers to create much more complex enclaves.

Further, building more efficient protocols, not only for off-chain computation in blockchains or machine-learning inference, will be crucial to increase adoption. For blockchains, we will see more complex smart contracts and potentially academic research on securely using peripheral devices (smart oracles) with a TEE. For machine learning, it will be essential to support federated learning within a heterogeneous cluster of TEEs (e.g., the TEEs on mobile devices and TEEs on servers).

In addition to these topics, microarchitectural attacks remain a persistent threat, even to memory-corruption defenses [34]. In the future, we will likely see more comprehensive defenses that span from the architectural to the microarchitectural level. These defenses must also be more flexible, as microarchitectural mitigations can degrade performance significantly while the risk of being the target of such attacks might be tolerable in certain scenarios, e.g., when ensuring that a physical core on a machine is never shared with another tenant in a cloud setup. As a result, the ability to configure defenses based on high-level, per-case security policies will be necessary.

7

List of Own Publications

7.1 Peer-Reviewed Publications

Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: Survival of the Fittest Fuzzing Mutators. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 24-27, 2023*, 2023. CORE Rank A*. **Distinguished Paper Award**.

Tommaso Frassetto, Patrick Jauernig, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi. POSE: Practical Off-chain Smart Contract Execution. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 24-27, 2023*, 2023. CORE Rank A*.

Jianqiang Wang, Pouya Mahmoody, Ferdinand Brasser, Patrick Jauernig, Ahmad-Reza Sadeghi, Donghui Yu, Dahan Pan, and Yuanyuan Zhang. Virtee: A Full Backward-Compatible TEE with Native Live Migration and Secure I/O. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, pages 241–246, New York, NY, USA, August 2022. Association for Computing Machinery. CORE Rank A.

Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. Riscyrop: Automated Return-Oriented Programming Attacks on RISC-V and Arm64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, pages 30–42, New York, NY, USA, October 2022. Association for Computing Machinery. CORE Rank A.

David Koisser, Patrick Jauernig, Gene Tsudik, and Ahmad-Reza Sadeghi. V'CER: Efficient certificate validation in constrained networks. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association. CORE Rank A*.

Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. CFInsight: A comprehensive metric for CFI policies. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022. CORE Rank A*.

Martin Schonstedt, Ferdinand Brasser, Patrick Jauernig, Emmanuel Stapf, and Ahmad-Reza Sadeghi. SafeTEE: Combining safety and security on ARM-based microcontrollers. In *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022*. IEEE, March 2022. CORE Rank B.

Aakash Tyagi, Addison Crump, Ahmad-Reza Sadeghi, Garrett Persyn, Jeyavijayan Rajendran, Patrick Jauernig, and Rahul Kande. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association. CORE Rank A*.

Tigist Abera, Ferdinand Brasser, Lachlan J. Gunn, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. Granddetauto: Detecting Malicious Nodes in Large-Scale Autonomous Networks. In Leyla Bilge and Tudor Dumitras, editors, *RAID '21: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 6-8, 2021*, pages 220–234. ACM, 2021. CORE Rank A.

Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, A. Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021. CORE Rank A*.

Sebastian P. Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. Offline Model Guard: Secure and Private ML on Mobile Devices. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 460–465. IEEE, 2020. https://ieeexplore.ieee.org/document/9116560. CORE Rank B.

P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A. Sadeghi. Fastkitten: Practical Smart Contracts on Bitcoin. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 801–818. USENIX Association, 2019. CORE Rank A*.

Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. CORE Rank A*. **Top Picks in Hardware and Embedded Security, 2021. Patent registered**.

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. CORE Rank A*.

Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 83–97. USENIX Association, 2018. CORE Rank A*.

7.2 Invited Publications & Technical Reports

Ferdinand Brasser, Patrick Jauernig, Frederik Pustelnik, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted Container Extensions for Container-Based Confidential Computing. *arXiv e-prints*, page arXiv:2205.05747, May 2022.

Emmanuel Stapf, Patrick Jauernig, Ferdinand Brasser, and Ahmad-Reza Sadeghi. In Hardware We Trust? From TPM to Enclave Computing on RISC-V. In *29th IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2021, Singapore, Singapore, October 4-7, 2021*, pages 1–6. IEEE, 2021.

Ghada Dessouky, Patrick Jauernig, Nele Mentens, Ahmad-Reza Sadeghi, and Emmanuel Stapf. INVITED: AI utopia or dystopia - on securing AI platforms. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*, pages 1–6. IEEE, 2020.

Lejla Batina, Patrick Jauernig, Nele Mentens, Ahmad-Reza Sadeghi, and Emmanuel Stapf. In Hardware We Trust: Gains and Pains of Hardware-Assisted Security. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 44. ACM, 2019.

7.3 Magazine Articles & Books

Ferdinand Brasser, Anrin Chakraborti, Reza Curtmola, Patrick Jauernig, Jonathan Katz, Jason Nieh, Ahmad-Reza Sadeghi, Radu Sion, Emmanuel Stapf, and Yinqian Zhang. *Cloud Computing Security: Foundations and Research Directions*. now, 2022.

Ghada Dessouky, Tommaso Frassetto, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. With Great Complexity Comes Great Vulnerability: From Stand-Alone Fixes to Reconfigurable Security. *IEEE Secur. Priv.*, 18(5):57–66, 2020.

Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted Execution Environments: Properties, Applications, and Challenges. *IEEE Secur. Priv.*, 18(2):56–60, 2020.

7.4 Posters

Sebastian P. Bayerl, Ferdinand Brasser, Christoph Busch, Tommaso Frassetto, Patrick Jauernig, Jascha Kolberg, Andreas Nautsch, Korbinian Riedhammer, Ahmad-Reza Sadeghi, and Thomas Schneider. Privacy-Preserving Speech Processing Via STPC and TEEs. *ACM CCS Workshop onPrivacy Preserving Machine Learning (PPML)*, 2019.

Bibliography

[1] Daniel Aarno and Jakob Engblom. *Software and System Development Using Virtual Platforms: Full-System Simulation with Wind River Simics*. Morgan Kaufmann, 2014.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, et al. Tensorflow: A System for Large-Scale Machine Learning. In *USENIX OSDI*, pages 265–283, 2016.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security*, 13(1):1–40, November 2009.

[4] Tigist Abera, Ferdinand Brasser, Lachlan J. Gunn, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. Granddetauto: Detecting Malicious Nodes in Large-Scale Autonomous Networks. In Leyla Bilge and Tudor Dumitras, editors, *RAID '21: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 6-8, 2021*, pages 220–234. ACM, 2021.

[5] Shimaa Ahmed, Amrita Roy Chowdhury, Kassem Fawaz, and Parmesh Ramanathan. Preech: A System for Privacy-Preserving Speech Transcription. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2703–2720. USENIX Association, 2020.

[6] AMD. AMD Secure Encrypted Virtualization (SEV).

[7] AMD. Strengthening Vm Isolation with Integrity Protection and More. *White Paper, January*, 2020.

[8] Anyl. Anyledger-Wallet. GitHub, 2019.

[9] ARM Limited. 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.

[10] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {SCONE}: Secure Linux containers with Intel {SGX}. In *12th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 16), volume 16, pages 689–703, 2016.

[11] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[12] Brandon Azad. Project Zero: Examining Pointer Authentication on the iPhone XS, February 2019.

[13] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, A. Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[14] Mauro Barni, Pierluigi Failla, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Privacy-Preserving ECG Classification With Branching Programs and Neural Networks. *IEEE Transactions on Information Forensics and Security*, 6(2):452–468, June 2011.

[15] Lejla Batina, Patrick Jauernig, Nele Mentens, Ahmad-Reza Sadeghi, and Emmanuel Stapf. In Hardware We Trust: Gains and Pains of Hardware-Assisted Security. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 44. ACM, 2019.

[16] Sebastian P. Bayerl, Ferdinand Brasser, Christoph Busch, Tommaso Frassetto, Patrick Jauernig, Jascha Kolberg, Andreas Nautsch, Korbinian Riedhammer, Ahmad-Reza Sadeghi, and Thomas Schneider. Privacy-Preserving Speech Processing Via STPC and TEEs. *ACM CCS Workshop onPrivacy Preserving Machine Learning (PPML)*, 2019.

[17] Sebastian P. Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. Offline Model Guard: Secure and Private ML on Mobile Devices. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 460–465. IEEE, 2020. https://ieeexplore.ieee.org/document/9116560.

[18] Chong Xu Bing Sun, Jin Liu. How to Survive the Hardware Assisted Control-Flow Integrity Enforcement (Black Hat Asia 2019) - InfoconDB. Black Hat Asia.

[19] Kyle Bittner, Martine De Cock, and Rafael Dowsley. Private Speech Classification with Secure Multiparty Computation. *CoRR*, abs/2007.00253, July 2020.

arXiv:2007.00253 [cs] type: article.

[20] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[21] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-Based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[22] Andy Boxall. Google Pixel 5 Review: Google's Best in a Compact Package, October 2020.

[23] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016.

[24] Jacob Bramley, Dejice Jacob, Andrei Lascu, Jeremy Singer, and Laurence Tratt. Picking a CHERI Allocator: Security and Performance Considerations.

[25] F. Brasser, B. El Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: Tiny Trust Anchor for Tiny Devices. In *DAC*, pages 1–6. IEEE, 2015.

[26] F. Brasser, T. Frassetto, K. Riedhammer, A. Sadeghi, T. Schneider, and C. Weinert. Voiceguard: Secure and Private Speech Processing. In B. Yegnanarayana, editor, *Interspeech*, pages 1303–1307. ISCA, 2018.

[27] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: automated and adjustable side-channel protection for SGX using data location randomization. In David Balenson, editor, *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, pages 788–800. ACM, 2019.

[28] Ferdinand Brasser, Anrin Chakraborti, Reza Curtmola, Patrick Jauernig, Jonathan Katz, Jason Nieh, Ahmad-Reza Sadeghi, Radu Sion, Emmanuel Stapf, and Yinqian Zhang. *Cloud Computing Security: Foundations and Research Directions*. now, 2022.

[29] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[30] Ferdinand Brasser, Patrick Jauernig, Frederik Pustelnik, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted Container Extensions for Container-Based Confidential Computing. *arXiv e-prints*, page arXiv:2205.05747, May 2022.

[31] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 50(1):1–33, April 2017.

[32] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 985–999. IEEE, 2019.

[33] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep Reinforcement Fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, pages 116–122, San Francisco, CA, USA, 2018. IEEE. Literaturangaben.

[34] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking Data on Meltdown-Resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019.

[35] Scott A. Carr and Mathias Payer. Datashield: Configurable Data Confidentiality and Integrity. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 193–204. ACM, 2017.

[36] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 147–160. USENIX Association, 2006.

[37] Chen Chen, Rahul Kande, Nathan Nyugen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HyPFuzz: Formal-Assisted Processor Fuzzing. *arXiv*, April 2023. arXiv:2304.02485 [cs] type: article.

[38] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[39] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and*

Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008, ASPLOS XIII, pages 2–13, New York, NY, USA, March 2008. ACM.

[40] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, May 2016. ISSN: 2375-1207.

[41] 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, Stockholm, Sweden, jun 2019. IEEE. Literaturangaben.

[42] Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. Riscyrop: Automated Return-Oriented Programming Attacks on RISC-V and Arm64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, pages 30–42, New York, NY, USA, October 2022. Association for Computing Machinery.

[43] Tobias Cloosters, Michael Rodler, and Lucas Davi. TEErex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 841–858, 2020.

[44] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. {SGXFuzz}: Efficiently Synthesizing Nested Structures for {SGX} Enclave Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3147–3164, 2022.

[45] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-Based Memory Isolation Systems. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1409–1426, USA, August 2020. USENIX Association. Literaturangaben.

[46] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, New York, NY, USA, October 2015. Association for Computing Machinery.

[47] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. Selfrando: Securing the Tor Browser against De-Anonymization Exploits. *Proceedings on Privacy Enhancing Technologies*, 2016(4):454–469, jul 2016.

[48] Victor Costan and Srinivas Devadas. Intel SGX Explained. *Cryptology ePrint Archive*, 2016.

[49] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 857–874. USENIX Association, 2016.

[50] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure, May 2015.

[51] Dan Rosenberg. Reflections on Trusting TrustZone. `https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf`, 2014.

[52] P. Das, L. Eckey, T. Frassetto, D. Gens, K. Hostáková, P. Jauernig, S. Faust, and A. Sadeghi. Fastkitten: Practical Smart Contracts on Bitcoin. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 801–818. USENIX Association, 2019.

[53] Leila Delshadtehrani, Sadullah Canakci, Manuel Egele, and Ajay Joshi. SealPK: Sealable Protection Keys for RISC-V. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1278–1281, February 2021. ISSN: 1558-1101.

[54] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. Phmon: A Programmable Hardware Monitor and Its Security Use Cases. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 807–824, 2020.

[55] Liang Deng, Qingkai Zeng, and Yao Liu. Isboxing: An Instruction Substitution Based Data Sandboxing for x86 Untrusted Libraries. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 386–400. Springer, 2015.

[56] Ghada Dessouky, Tommaso Frassetto, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. With Great Complexity Comes Great Vulnerability: From Stand-Alone Fixes to Reconfigurable Security. *IEEE Secur. Priv.*, 18(5):57–66, 2020.

[57] Ghada Dessouky, Patrick Jauernig, Nele Mentens, Ahmad-Reza Sadeghi, and Emmanuel Stapf. INVITED: AI utopia or dystopia - on securing AI platforms. In

57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020, pages 1–6. IEEE, 2020.

[58] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight Jr., Benjamin C. Pierce, and André DeHon. Architectural Support for Software-Defined Metadata Processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 487–502. ACM, 2015.

[59] Zhen Yu Ding and Claire Le Goues. An Empirical Study of Oss-Fuzz Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 131–142. IEEE, 2021.

[60] William Drozd and Michael D. Wagner. Fuzzergym: A Competitive Framework for Fuzzing and Learning. *arXiv e-prints*, page arXiv:1807.07490, July 2018.

[61] Nir Drucker and Shay Gueron. Combining Homomorphic Encryption with Trusted Execution Environment: A Demonstration with Paillier Encryption and SGX. In *Proceedings of the 2017 international workshop on managing insider security threats*, pages 85–88, 2017.

[62] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual Payment Hubs Over Cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 106–123. IEEE, 2019.

[63] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. The Untapped Potential of Trusted Execution Environments on Mobile Devices. *IEEE Secur. Priv.*, 12(4):29–37, 2014.

[64] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: secure and minimal architecture for (establishing dynamic) root of trust. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[65] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *2015 IEEE Symposium on Security and Privacy*, pages 781–796. IEEE, May 2015. ISSN: 2375-1207.

[66] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable Memory Protection in the PENGLAI Enclave. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on*

Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021, pages 275–294. USENIX Association, 2021.

[67] Oasis Protocol Foundation. Oasis Network.

[68] Tommaso Frassetto, Patrick Jauernig, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi. POSE: Practical Off-chain Smart Contract Execution. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 24-27, 2023*, 2023.

[69] Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. CFInsight: A comprehensive metric for CFI policies. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.

[70] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 83–97. USENIX Association, 2018.

[71] Gal Beniamini. Qsee Privilege Escalation Vulnerabilitiy. `http://bits-please.blogspot.de/2015/08/full-trustzone-exploit-for-msm8974.html`, 2015.

[72] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. Greyone: Data Flow Sensitive Fuzzing. In *USENIX Security Symposium*, pages 2577–2594, 2020.

[73] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 201–210. JMLR.org, 2016.

[74] Google. Afl - American Fuzzy Lop, April 2023. original-date: 2019-07-25T16:50:06Z.

[75] Richard Grisenthwaite. Arm Morello Evaluation Platform-Validating CHERI-Based Security in a High-Performance System. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–22. IEEE Computer Society, 2022.

[76] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and Secure DNN Inference with Enclaves. *CoRR*, abs/1810.00602, September 2019. arXiv:1810.00602 [cs] type: article.

[77] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. TrustShadow: Secure execution of unmodified applications with ARM TrustZone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 488–501, New York, NY, USA, June 2017. ACM.

[78] Lachlan J. Gunn, Jian Liu, Bruno Vavala, and N. Asokan. Making Speculative BFT Resilient with Trusted Monotonic Counters. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, October 2019.

[79] Nikolaus Hansen, Dirk V Arnold, and Anne Auger. Evolution Strategies. *Springer handbook of computational intelligence*, pages 871–898, 2015.

[80] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Maximilian Augustin, Michael Backes, and Mario Fritz. MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 3295–3304, Nashville, TN, USA, jun 2021. IEEE.

[81] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. Partemu: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 789–806, 2020.

[82] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. In Longbo Huang, Anshul Gandhi, Negar Kiyavash, and Jia Wang, editors, *SIGMETRICS '21: ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, Virtual Event, China, June 14-18, 2021*, pages 81–82. ACM, 2021.

[83] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 489–504. USENIX Association, 2019.

[84] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: Secure Applications on an Untrusted Operating System. In Vivek Sarkar and Rastislav Bodík, editors, *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '13*, pages 265–278. ACM, 2013.

[85] Chenlin Huang, Yusong Tan, Guoyun Duan, Zhiwen Chen, Boyang Zhang, Peiyao Deng, Qianxiang Zhang, Jianhua Sun, Hao Chen, Guoqing Xiao, et al. A Coverage-

Guided Fuzzing Framework for Trusted Execution Environments. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 775–782. IEEE, 2021.

[86] Guerney Hunt, Richard (Rick) Boivie, Eric Hall, Elaine Palmer, Dimitrios Pendarakis, and Enriquillo (Ray) Valdez. Supporting Protected Computing on IBM Power Architecture.

[87] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure Computing with Cloud Gpus. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 817–833. USENIX Association, 2020.

[88] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-Preserving Machine Learning As a Service. *CoRR*, abs/1803.05961, 2018.

[89] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, IEEE, may 2021.

[90] Nick Hynes, Raymond Cheng, and Dawn Song. Efficient Deep Learning on Multi-Source Private Data. *CoRR*, abs/1807.06689, 2018.

[91] Hyperledger Foundation. Hyperledger Avalon.

[92] Synopsys Inc. Chromium (google Chrome), August 2019.

[93] Intel. Intel® Trust Domain Extensions.

[94] Intel. Intel Software Guard Extensions Programming Reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, 2014.

[95] Intel. Intel® eASIC™ Devices, 2023.

[96] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM*

SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, pages 1868–1882. ACM, 2018.

[97] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: Survival of the Fittest Fuzzing Mutators. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 24-27, 2023*, 2023.

[98] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted Execution Environments: Properties, Applications, and Challenges. *IEEE Secur. Priv.*, 18(2):56–60, 2020.

[99] Patrick Thomas Jauernig. SMOV: Lighweight In-Process Memory Isolation. Master's thesis, Darmstadt, November 2017.

[100] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1651–1669. USENIX Association, 2018.

[101] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, Private Smart Contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.

[102] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. `https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`, 2016.

[103] David Kaplan. Protecting Vm Register State with SEV-ES. *White paper*, 2017.

[104] The kernel development community. Memory Protection Keys.

[105] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[106] P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan. Trustlite: A Security Architecture for Tiny Embedded Devices. In *EuroSys*, page 10. ACM, 2014.

[107] David Koisser, Patrick Jauernig, Gene Tsudik, and Ahmad-Reza Sadeghi. V'CER: Efficient certificate validation in constrained networks. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[108] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings*

of the Twelfth European Conference on Computer Systems, EuroSys '17, pages 437–452, New York, NY, USA, April 2017. ACM.

[109] Ranjit Kumaresan and Iddo Bentov. Amortizing Secure Computation with Penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 418–429, 2016.

[110] 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*, pages 195–206, 2015.

[111] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to Secure Computation with Penalties. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 406–417, 2016.

[112] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In Per Larsen and Ahmad-Reza Sadeghi, editors, *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. ACM / Morgan & Claypool, March 2018.

[113] Michael Larabel. Linux 5.12 Coming In At Around 28.8 Million Lines, AMDGPU Driver Closing In On 3 Million, 2021.

[114] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[115] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-Preserving Remote Deep-Learning Inference Using SGX. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, pages 1–17, New York, NY, USA, October 2019. Association for Computing Machinery.

[116] Caroline Lemieux and Koushik Sen. Fairfuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[117] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A Two-Way Sandbox for x86 Native Code. In Garth Gibson and Nickolai Zeldovich, editors, *USENIX ATC*, pages 409–420. USENIX Association, 2014.

[118] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the*

2017 11th Joint Meeting on Foundations of Software Engineering, pages 627–637, 2017.

[119] Guangcheng Liang, Lejian Liao, Xin Xu, Jianguang Du, Guoqiang Li, and Henglong Zhao. Effective Fuzzing Based on Dynamic Taint Analysis. In *2013 Ninth International Conference on Computational Intelligence and Security*, pages 615–619. IEEE, 2013.

[120] Hans Liljestrand, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. Authenticated Call Stack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 1–2, New York, NY, USA, June 2019. Association for Computing Machinery.

[121] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. PACStack: An authenticated call stack. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 357–374. USENIX Association, 2021.

[122] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 177–194. USENIX Association, 2019.

[123] Linaro. OP-TEE. `https://www.op-tee.org/`.

[124] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 49–64. USENIX Association, 2016.

[125] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious Neural Network Predictions Via Minionn Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 619–631, New York, NY, USA, October 2017. Association for Computing Machinery.

[126] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable Byzantine Consensus Via Hardware-Assisted Secret Sharing. *IEEE Transactions on Computers*, 68(1):139–151, January 2019.

[127] Renju Liu, Luis Garcia, Zaoxing Liu, Botong Ou, and Mani B. Srivastava. Secdeep: Secure and Performant On-Device Deep Learning Inference Framework for Mobile and Iot Devices. In *IoTDI '21: International Conference on Internet-of-Things Design*

and Implementation, Virtual Event / Charlottesville, VA, USA, May 18-21, 2021, pages 67–79. ACM, 2021.

[128] Arm Ltd. Arm Confidential Compute Architecture.

[129] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, pages 1949–1966, 2019.

[130] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. Trustvisor: Efficient TCB Reduction and Attestation. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 143–158, Oakland, CA, USA, 2010. IEEE Computer Society.

[131] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[132] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment Channels That Go Faster Than Lightning. *CoRR, abs/1702.05812*, 2017.

[133] Matt Miller. Trends, Challenge, and Shifts in Software Vulnerability Mitigation, January 2019. original-date: 2017-07-06T21:55:08Z.

[134] MITRE. Log4shell Cve Details.

[135] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. Microstache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*, volume 11050 of *Lecture Notes in Computer Science*, pages 359–379. Springer, 2018.

[136] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, San Jose, CA, USA, may 2017. IEEE. Literaturangaben.

[137] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. SEVered: Subverting AMD's virtual machine encryption. In *EuroSec*. ACM, 2018.

[138] Deepika Natarajan, Andrew Loveless, Wei Dai, and Ronald Dreslinski. Chex-Mix: Combining Homomorphic Encryption with Trusted Execution Environments for

Oblivious Inference in the Cloud. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE, July 2023.

[139] Nicole Nichols, Mark Raugas, Robert Jasper, and Nathan Hilliard. Faster Fuzzing: Reinitialization with Deep Neural Models. *arXiv preprint arXiv:1711.02807*, 2017.

[140] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-Cost Trustworthy Extensible Networked Devices with a Zero-Software Trusted Computing Base. In *USENIX Security*, 2013.

[141] J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A Low-Cost Security Architecture for Iot Devices. *TOPS*, 20(3):7, 2017.

[142] Satsuya Ohata. Recent Advances in Practical Secure Multi-Party Computation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 103(10):1134–1141, 2020.

[143] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 619–636. USENIX Association, 2016.

[144] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):28:1–28:30, June 2018.

[145] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting {SGX} Enclaves from Practical Side-Channel Attacks. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 227–240, 2018.

[146] 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.

[147] Claudio Orlandi, Alessandro Piva, and Mauro Barni. Oblivious Neural Network Computing Via Homomorphic Encryption. *EURASIP J. Inf. Secur.*, 2007, 2007.

[148] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing Os Fuzzer Seed Selection with Trace Distillation. In William Enck and Adrienne Porter

Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 729–743. USENIX Association, 2018.

[149] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software Abstraction for Intel Memory Protection Keys (intel MPK). In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 241–254. USENIX Association, 2019.

[150] Manas A. Pathak, Bhiksha Raj, Shantanu D. Rane, and Paris Smaragdis. Privacy-Preserving Speech Processing: Cryptographic and String-Matching Frameworks Show Promise. *IEEE Signal Processing Magazine*, 30(2):62–74, March 2013.

[151] Travis Patron. What's the Big Idea behind Ethereum's World Computer. `https://www.coindesk.com/whats-big-idea-behind-ethereums-world-computer/`.

[152] Douglas Perry. Linux Kernel Grows Past 15 Million Lines of Code, January 2012.

[153] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2019.

[154] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-Preserving Video Analytics As a Cloud Service. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1039–1056. USENIX Association, 2020.

[155] Joseph Poon and Vitalik Buterin. Plasma: Scalable Autonomous Smart Contracts. *White paper*, pages 1–47, 2017.

[156] Project Zero. Trust Issues: Exploiting TrustZone TEEs. `https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html`, 2017.

[157] Mohit Rajpal, William Blum, and Rishabh Singh. Not All Bytes Are Equal: Neural Byte Sieve for Fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[158] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-Aware Evolutionary Fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[159] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi. Appsec: A Safe Execution Environment for Security Sensitive Applications. In Ada Gavrilovska, Angela Demke Brown, and Bjarne Steensgaard, editors, *Proceedings of the 11th ACM SIGPLAN/SIGOPS*

International Conference on Virtual Execution Environments, pages 187–199. ACM, March 2015.

[160] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: xnor-based oblivious deep neural network inference. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1501–1518. USENIX Association, 2019.

[161] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 707–721, New York, NY, USA, May 2018. Association for Computing Machinery.

[162] Ahmad-Reza Sadeghi and Thomas Schneider. Generalized Universal Circuits for Secure Evaluation of Private Functions with Application to Data Classification. In *Information Security and Cryptology – ICISC 2008*, volume 2008, page 453. Springer Berlin Heidelberg, December 2008.

[163] Alexander Schlögl and Rainer Böhme. Ennclave: Offline Inference with Model Confidentiality. In Jay Ligatti and Xinming Ou, editors, *AISec@CCS 2020: Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security, Virtual Event, USA, 13 November 2020*, pages 93–104. ACM, 2020.

[164] Martin Schonstedt, Ferdinand Brasser, Patrick Jauernig, Emmanuel Stapf, and Ahmad-Reza Sadeghi. SafeTEE: Combining safety and security on ARM-based microcontrollers. In *2022 Design, Automation & Test in Europe Conference & Exhibition, DATE 2022*. IEEE, March 2022.

[165] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruß. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86: 29th USENIX Security Symposium. In *29th USENIX Security Symposium (USENIX Security 20)*, Proceedings of the 29th USENIX Security Symposium, pages 1677–1694. USENIX Association, August 2020.

[166] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications, May 2015.

[167] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 1–12. USENIX Association, 2010.

[168] Gabriele Serra, Pietro Fara, Giorgiomaria Cicero, Francesco Restuccia, and Alessandro Biondi. PAC-Pl: Enabling control-flow integrity with pointer authentication in FPGA SoC platforms, 2022.

[169] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–11. ACM, June 2019.

[170] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications with SGX Enclaves. In *NDSS*, 2017.

[171] Lingyun Situ, Linzhang Wang, Xuandong Li, Le Guan, Wenhui Zhang, and Peng Liu. Energy Distribution Matters in Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 270–271. IEEE, 2019.

[172] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization, 2013.

[173] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *S&P*, pages 1–17. IEEE, IEEE, May 2016.

[174] Emmanuel Stapf, Patrick Jauernig, Ferdinand Brasser, and Ahmad-Reza Sadeghi. In Hardware We Trust? From TPM to Enclave Computing on RISC-V. In *29th IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2021, Singapore, Singapore, October 4-7, 2021*, pages 1–6. IEEE, 2021.

[175] Statista. Most Used Languages among Software Developers Globally 2022.

[176] Stefan Steinegger, David Schrammel, Samuel Weiser, Pascal Nasahl, and Stefan Mangard. Servas! Secure Enclaves Via RISC-V Authenticryption Shield. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part II*, volume 12973 of *Lecture Notes in Computer Science*, pages 370–391. Springer, 2021.

[177] Nick Stephens. Behind the Pwn of a TrustZone. `https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose`, 2016.

[178] Nenad Stojanovski, Marjan Gusev, Danilo Gligoroski, and Svein J Knapskog. Bypassing Data Execution Prevention on Microsoftwindows Xp Sp2. In *The Second*

International Conference on Availability, Reliability and Security (ARES'07), pages 1222–1226. IEEE, 2007.

[179] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems, 2010.

[180] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. TrustICE: Hardware-assisted isolated computing environments on mobile devices. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, June 2015.

[181] Zhichuang Sun, Ruimin Sun, Long Lu, and Somesh Jha. Shadownet: A Secure and Efficient System for On-Device Model Inference. *CoRR*, abs/2011.05905, 2020.

[182] Synopsys. Heartbleed, 2022.

[183] The Linux Foundation [@linuxfoundation]. The Kernel Right Now Is about 35 Million Lines of Code, June 2020.

[184] Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain Scaling Using Rollups: A Comprehensive Survey. *IEEE Access*, 2022.

[185] Florian Tramèr and Dan Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[186] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing Hardware like Software. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3237–3254, 2022.

[187] C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library Os for Unmodified Applications on SGX. In *USENIX ATC*, pages 645–658, 2017.

[188] Aakash Tyagi, Addison Crump, Ahmad-Reza Sadeghi, Garrett Persyn, Jeyavijayan Rajendran, Patrick Jauernig, and Rahul Kande. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[189] Anjo Vahldiek-Oberwagner, Eslam Elnikety, N. Duarte, Michael Sammler, P. Druschel, and D. Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*, 2019.

[190] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-Of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[191] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. `https://sgaxeattack.com/`, 2020.

[192] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on Gpus. In *USENIX OSDI 18*, pages 681–696, Carlsbad, CA, October 2018. USENIX Association.

[193] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In Andrew P. Black and Barbara Liskov, editors, *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993*, pages 203–216. ACM, 1993.

[194] DanChen Wang, Xiaosong Zhang, Yang Xu, and Haiquan Song. A Secure Multi-Party Computing System Based on SGX Technology for Trusted Data Circulation. In *2019 IEEE 14th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*, pages 90–95. IEEE, 2019.

[195] Gang Wang and Mark Nixon. Sok: X-Assisted Bft Consensus Protocols. In *Blockchain – ICBC 2023*, pages 54–71. Springer Nature Switzerland, 2023.

[196] Jianqiang Wang, Pouya Mahmoody, Ferdinand Brasser, Patrick Jauernig, Ahmad-Reza Sadeghi, Donghui Yu, Dahan Pan, and Yuanyuan Zhang. Virtee: A Full Backward-Compatible TEE with Native Live Migration and Secure I/O. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, pages 241–246, New York, NY, USA, August 2022. Association for Computing Machinery.

[197] Wenhao Wang, Yichen Jiang, Qintao Shen, Weihao Huang, Hao Chen, Shuang Wang, XiaoFeng Wang, Haixu Tang, Kai Chen, Kristin Lauter, et al. Toward Scalable Fully Homomorphic Encryption through Light Trusted Computing Assistance. *arXiv preprint arXiv:1905.07766*, 2019.

[198] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. Neufuzz: Efficient Fuzzing with Deep Neural Network. *IEEE Access*, 7:36340–36352, 2019.

[199] Pete Warden. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv preprint arXiv:1804.03209*, 2018.

[200] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating Code from Data in x86 Binaries. In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, editors, *Machine Learning and Knowledge Discovery in Databases*, Lecture Notes in Computer Science, pages 522–536, Berlin, Heidelberg, 2011. Springer.

[201] Mario Werner, Thomas Unterluggauer, Robert Schilling, David Schaffenrath, and Stefan Mangard. Transparent Memory Encryption and Authentication. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Ghent, Belgium, sep 2017. IEEE.

[202] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, IEEE, June 2014.

[203] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One Fuzzing Strategy to Rule Them All. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, May 2022.

[204] Pengfei Wu, Jianting Ning, Jiamin Shen, Hongbing Wang, and Ee-Chien Chang. Hybrid Trust Multi-Party Computation with Trusted Execution Environment. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.

[205] Karl Wüst, Loris Diana, Kari Kostiainen, Ghassan Karame, Sinisa Matetic, and Srdjan Capkun. Bitcontracts: Adding Expressive Smart Contracts to Legacy Cryptocurrencies. *IACR Cryptol. ePrint Arch.*, 2019:857, 2019.

[206] Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. Ace: Asynchronous and Concurrent Execution of Complex Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 587–600, 2020.

[207] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692, May 2020.

[208] Jisoo Yang and Kang G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis. In David Gregg, Vikram S. Adve, and Brian N. Bershad, editors, *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008*, pages 71–80. ACM, 2008.

[209] Timothy Yang, Galen Andrew, Hubert Eichner, Haicheng Sun, Wei Li, Nicholas Kong, Daniel Ramage, and Françoise Beaufays. Applied Federated Learning: Improving Google Keyboard Query Suggestions. *arXiv preprint arXiv:1812.02903*, 2018.

[210] Joseph Yiu. ARMv8-m architecture technical overview. *ARM white paper*, 2015.

[211] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive Energy-Saving Greybox Fuzzing As a Variant of the Adversarial Multi-Armed Bandit. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 2307–2324, 2020.

[212] Google Projekt Zero. Xnu: Copy-On-Write Behavior Bypass Via Mount of User-Owned Filesystem Image. `https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`, 2018.

[213] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. Truspy: Cache Side-Channel Information Leakage from the Secure World on Arm Devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[214] Ning Zhang, Kun Sun, Wenjing Lou, and Y. Thomas Hou. Case: Cache-Assisted Secure Execution on Arm Processors, 2016.

[215] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. SecTEE: A Software-Based Approach to Secure Enclave Architecture Using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, November 2019.

[216] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: Efficient Protected Shadow Stacks for Embedded Systems. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1219–1236. USENIX Association, 2020.

[217] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-Based Fault Isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 558–569. ACM, 2014.

[218] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-Fat: Architectural Support for Low Overhead Memory Safety Checks. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*, pages 916–929. IEEE, 2021.

[219] Serkan Özkan. Browse Vulnerabilities by Date, 2022.

List of Figures

Acronyms

CCA Confidential Compute Architecture

CET Control-flow Enforcement Technology

CFI Control-Flow Integrity

CPI Code Pointer Integrity

DMA Direct Memory Access

ES Evolution Strategy

FPGA Field Programmable Gate Array

ISA Instruction Set Architecture

MMU Memory Management Unit

MPX Memory Protection Extensions

MPK Memory Protection Keys

PKU Protection Keys for Userspace

PSO Particle Swarm Optimization

SEV Secure Encrypted Virtualization

SFI Software-fault Isolation

SGX Software Guard Extensions

TCB Trusted Computing Base

TDX Trust Domain Extensions

TEE Trusted Execution Environment

APPENDICES

A

IMIX: In-Process Memory Isolation EXtension (USENIX Sec'18)

---

[70] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 83–97. USENIX Association, 2018. CORE Rank A*. Chapter 2.

IMIX: In-Process Memory Isolation EXtension

Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi, *Technische Universität Darmstadt*

https://www.usenix.org/conference/usenixsecurity18/presentation/frassetto

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

IMIX: In-Process Memory Isolation EXtension

Tommaso Frassetto Patrick Jauernig Christopher Liebchen Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
{tommaso.frassetto, patrick.jauernig, christopher.liebchen, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

Memory-corruption attacks have been subject to extensive research in the latest decades. Researchers demonstrated sophisticated attack techniques, such as (just-in-time/blind) return-oriented programming and counterfeit object-oriented programming, which enable the attacker to execute arbitrary code and data-oriented attacks that are commonly used for privilege escalation. At the same time, the research community proposed a number of effective defense techniques. In particular, control-flow integrity (CFI), code-pointer integrity (CPI), and fine-grained code randomization are effective mitigation techniques against code-reuse attacks. All of these techniques require strong memory isolation. For example, CFI's shadow stack, CPI's safe-region, and the randomization secret must be protected from adversaries able to perform arbitrary read-write accesses.

In this paper we propose IMIX, a lightweight, in-process memory isolation extension for the Intel-based x86 CPUs. Our solution extends the x86 ISA with a new memory-access permission to mark memory pages as *security sensitive*. These memory pages can then only be accessed with a newly introduced instruction. Unlike previous work, IMIX is not tailored towards a specific defense (technique) but can be leveraged as a primitive to protect the data of a wide variety of memory-corruption defenses. We provide a proof of concept of IMIX using Intel's Simulation and Analysis Engine. We extend Clang/LLVM to include our new instruction, and enhance CPI by protecting CPI's safe region using IMIX.

1 Introduction

Memory-corruption attacks have been a major threat against modern software for multiple decades. Attackers leverage memory-corruption vulnerabilities to perform multiple malicious activities including taking control of systems and exfiltrating information. Memory-corruption attacks can be roughly divided into the categories code-injection [3], code-reuse [50, 52, 54], and data-only attacks [12, 28, 29]. While code-injection attacks introduce new malicious code into the vulnerable program, code-reuse attacks reuse the existing code in an unintended way. Data-only attacks in turn aim to influence the program behavior by modifying crucial data variables, e.g., used in branching conditions.

Defenses against memory-corruption typically reduce the attack surface by preventing the adversary from corrupting part of the application's memory which is essential for a successful attack. Prominent examples include: W⊕X [44, 48] which prevents data from being executed, and hence, code-injection attacks; Control Flow Integrity (CFI) [1] and Code-Pointer Integrity (CPI) [38] which protect code pointers to prevent code-reuse attacks; and Data Flow Integrity (DFI) [2, 10] mitigating data-only attacks by restricting data access.

Some of these defenses can be implemented efficiently using mechanisms that reside entirely outside the underlying application process. For instance, the kernel configures W⊕X and the hardware enforces it. Hence, the adversary cannot tamper with this defense mechanism when exploiting a memory-corruption vulnerability in the application. However, using an external mechanism is not always feasible in practice due to high performance overhead. For instance, CFI requires run-time checks and a shadow stack [1, 9, 18], which is updated every time a function is invoked or returns. CPI requires run-time checks and a *safe region*, which contains metadata about the program's variables. The required code for these defenses can be efficiently protected when marked as read-only, just like the application code. However, as of today no architectural solution exists that protects the data region of these defenses from unintended/malicious accesses. This data cannot be stored outside of the process, e.g., in kernel memory, because accessing it would impose an impractical performance overhead due to the time needed for a context switch. Hence, to pre-

vent the adversary from accessing the data some form of *in-process memory isolation* is needed, i.e., a mechanism ensuring access only by the defense code while denying access by the potentially vulnerable application code. However, devising a memory isolation scheme for current x86 processors is challenging.

Memory Isolation Approaches. A variety of memory isolation solutions have been proposed or deployed both in software and/or hardware. Software solutions use either access instrumentation [8, 61], or data hiding [6, 38]. Instrumentation-based memory isolation inserts run-time checks before every memory access in the untrusted code in order to prevent accesses to the protected region. However, it imposes a substantial performance overhead, for instance, code instrumented using Software Fault Isolation (SFI) incurs an overhead up to 43% [51]. Data hiding schemes typically allocate data at secret random addresses. Modern processors have sufficiently large virtual memory space (140 TB) to prevent brute-force attacks. The randomized base address must be kept secret and is usually stored in a CPU register. However, ensuring that this secret is not leaked to the adversary is challenging, especially if the program is very complex. For instance, compilers sometimes save registers to the stack in order to make room for intermediate results from some computation. This is known as *register spilling* and can leak the randomization secret [14]. Moreover, even a large address space can successfully be brute-forced as it was shown on an implementation of CPI [22, 24]. Thus, current in-process memory isolation either compromises performance or offers limited security.
Memory protection based on hardware extensions is another approach to achieve in-process isolation. For instance, Intel has recently announced Control-flow Enforcement Technology [33] and Memory Protection Keys [34] (already available on other architectures, e.g. *memory domains* on ARM32 [4]). However, these technologies either provide hardware support limited to a specific mitigation, or cause unnecessary performance overhead. We will discuss those technologies in a more detailed way in Section 8.

Goals and Contributions. In this paper we present IMIX, which enables lightweight in-process memory isolation for memory-corruption defenses that target the x86 architecture. IMIX enables *isolated pages*. Marked with a special flag, isolated pages can only be accessed using a single new instruction we introduce, called `smov`. Just like defenses like W⊕X protect the code of run-time defenses from unintended modifications, IMIX protects the data of those defenses from unintended access. In contrast to other recently proposed hardware-based approaches we provide an agnostic ISA extension that can be leveraged by a variety of defenses against code-reuse attacks to increase performance and security. To summarize, our main contributions are:

- **Hardware primitive to isolate data memory**. We propose IMIX, a novel instruction set architecture (ISA) extension to provide effective and efficient in-process isolation that is fundamental for the security of memory-corruption defenses (metadata protection). Therefore, IMIX introduces a new memory-access permission to protect the isolated pages, which prevents regular load and store instructions from accessing this memory. Instead, the code part of defense mechanisms needs to use our newly introduced `smov` instruction to access the protected data.
- **Proof-of-concept implementation**. We provide a fully-fledged proof of concept of IMIX. In particular, we leverage Intel's Simulation and Analysis Engine [11] to extend the x86 ISA with our new memory protection, and to add the *smov* instruction. Further, we extend the Linux kernel to support our ISA extension and the LLVM compiler infrastructure to provide primitives for allocation of protected memory, and access to the former. Finally, we demonstrate how defenses against memory-corruption attacks benefit from using IMIX by porting code-pointer integrity (CPI) [38] to leverage IMIX to isolate its safe-region.
- **Thorough evaluation**. We evaluate the performance by comparing our IMIX-enabled port of CPI to the original x86-64 variant. Further, we compare our solution to Intel's Memory Protection Keys and Intel's Memory Protection Extensions [34] overhead for CPI.

2 Background

In this section we provide the necessary technical background which is necessary for understanding the remainder of this paper. We first provide a brief summary of memory corruption attacks and defenses, and then explain memory protection on the x86 architecture.

2.1 Memory Corruption

C and C++ are popular programming languages due to their flexibility and efficiency. However, their requirement for manual memory management places a burden on developers, and mistakes easily result in memory-corruption vulnerabilities which enable attackers to change the behavior of a vulnerable application

during run time. For example, a missing bounds check during the access of a buffer can lead to a buffer overflow, which enables the attacker to manipulate adjacent memory values. With a write primitive in hand the attacker can achieve different levels of control of the target, such as changing data flows within the application, or hijacking the control flow. When conducting a data-flow attack [28, 29], the attacker manipulates data pointers and variables that are used in conditional statements to disclose secrets like cryptographic keys. In contrast, during a control-flow hijacking attack, the attacker overwrites code pointers, which are later used as a target address of an indirect branch, to change control flow to execute injected code [3] or to conduct a code-reuse attack [50, 52, 54].

There exist different approaches to mitigate these attacks, however, they all have in common that they are part of the same execution context as the vulnerable application, and often make a tradeoff between practicality and security.

For example, combining SoftBounds [46] and CETS [47] guarantees memory safety for applications written in C, and hence, prevent the exploitation of memory-corruption vulnerabilities in the first place. Unfortunately, these guarantees come with an impractical performance overhead of more than 100%. To limit the performance impact, other mitigation techniques focus on mitigating certain attack techniques. To mitigate control-flow hijacking attacks, these techniques prevent the corruption of code pointers [38], verify code pointers before they are used [1], or ensure that the values of valid code pointers are different for each execution [16].

Another common aspect of every memory-corruption mitigation technique is that they reduce the attack surface of a potentially vulnerable application to the mitigation itself. In other words, if the attacker is able to manipulate the mitigation or memory on which the mitigation depends, she can undermine the security of the mitigation. The protection mitigation's memory is hard because it is part of the memory which the attacker can potentially access.

Next, we provide a short overview memory protection techniques, which are available on the x86 architecture, that can be leveraged to protect the application's and mitigation's memory.

2.2 Memory Isolation

The x86 architecture offers different mechanisms to enforce memory protection. *Segmentation* and *paging* are the most well-known ones. However, recently, Intel and AMD proposed a number of additional features to protect and isolate memory. As we argue in Section 8, IMIX is most likely to be adapted for Intel-based x86 CPUs, hence, we focus in this section on memory protection features that are implemented or will be implemented for Intel-based x86 CPUs. Note that in most cases AMD provides a similar feature using different naming convention. Finally, we shortly discuss software-based memory isolation.

Traditional Memory Isolation. Segmentation and paging build a layer of indirection for memory accesses that can be configured by the operating system, and the CPU enforces access control while resolving the indirection.

Segmentation is a legacy feature that allows developers to define segments that consists of a start address, size, and an access permission. However, on modern 64-bit systems access permissions are no longer enforced. Nevertheless, many mitigations [6, 18, 38, 41] leverage segmentation to implement information hiding by allocating their data TCB at a random address, and ensure that it is only accessed through segmentation.

On modern systems, paging creates an indirection that maps *virtual memory* to *physical memory*. The mapping is configured by the operating system through a data structure known as *page tables*, which contain the translation information and a variety of access permissions. The paging permission system enables the operating system to assign memory to either itself or to the user mode. To isolate different processes from each other, the operating system ensures that each process uses its own page table. Due to legacy reasons, paging does not differentiate between the read and execute permission, which is why modern systems feature the "non-executable" permission. Further, paging allows to mark memory as (non-)writable.

New Memory Protection Features. Recently introduced or proposed features that enable memory isolation on x86 are Extended Page Tables (EPT), Memory Protection Extensions (MPX), Software Guard Extensions (SGX), Memory Protection Keys (MPK) and Control-flow Enforcement Technology (CET). We provide a comparison in Section 9.

The EPT facilitate memory virtualization and are conceptually the same as regular page tables, except that they are configured by the hypervisor, and allow to set the read/write/execute permission individually. Hence, previous work leveraged the EPT to implement execute-only memory [16, 58, 63]. MPX implements bounds checking in hardware. Therefore, it provides new instructions to configure a lower and upper bound for a pointer to a buffer. Then, before a pointer is dereferenced, the developer can leverage another MPX instruction to quickly check whether this address points into the buffers boundaries. SGX allows to create *enclaves* within a process

that are completely isolated from the rest of the system at the cost of high overhead when switching the execution to the code within an enclave. MPK introduces a new register, which contains a protection key, and enables programmers to tag memory (the tag is stored in the page table) such that it can only be accessed if the protection key register contains a specific key. MPK can be utilized to implement in-process isolation by tagging the security critical data and loading the corresponding key only when executing a benign access, and deleting it after the access succeeded. Intel's hardware support for CFI, CET, provides similar memory isolation the shadow stack as IMIX for security critical data in general. It introduces a new access permission for the shadow stack, and special instructions to access it. Unfortunately, CET is tailored towards CFI and cannot be easily repurposed for other mitigations.

Software-based Approaches. Software Fault Isolation (SFI) [43, 51, 61] instruments every read, write, and branch instruction to enable in-process isolation. However, this approach comes with a significant performance overhead due to the additional instructions.
To summarize, none of the above listed memory protection features provides mitigation-agnostic security and performance benefits at the same time.

3 Adversary Model

Throughout our work, we use the following standard adversary model and assumptions, which are consistent with prior work in this field of research [21, 38, 53, 54].

- **Memory corruption.** We assume the presence of a memory-corruption vulnerability, which the adversary can repeatedly exploit to read and write data according to the memory access permissions.

- **Sandboxed code execution.** The adversary can execute code in an isolated environment. However, the executed code cannot interfere with the target application by any means other than by using the memory corruption vulnerability. In particular, this means that the sandboxed code cannot execute the `smov` instruction with controlled arguments. Arbitrary code execution is prevented by hardening the target application with techniques such as CPI [38], CFI [1], or code randomization [16]. However, the attacker can target those defenses as well using the memory corruption vulnerability. We assume memory-corruption mitigations cannot be bypassed unless the attacker can corrupt the mitigation's metadata.

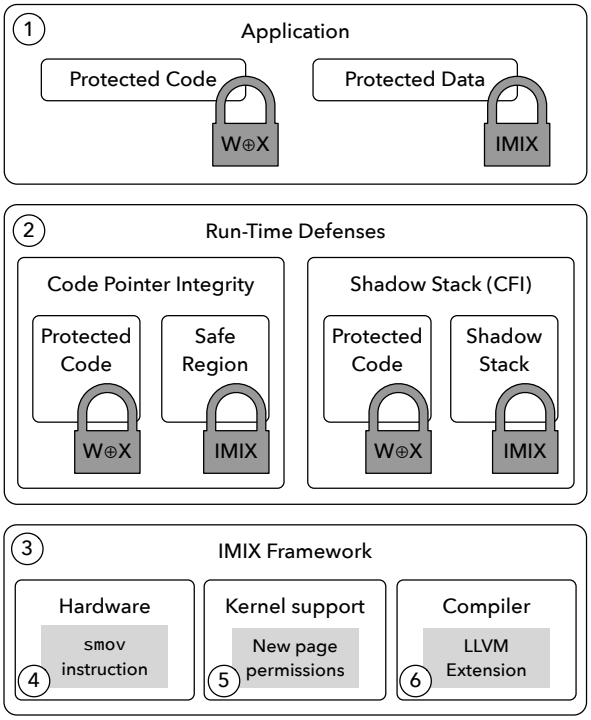


Figure 1: Overview of IMIX.

- **Immutable code.** The adversary cannot inject new code or modify existing code, which would allow her to execute the `smov` instruction with controlled arguments. This is enforced by hardening the target application with the W⊕X memory policy [44, 48].

4 IMIX

As we mentioned in Section 1, application developers protect their applications (① in Figure 1) using run-time defenses ②. Like for applications, the correct functionality of defenses relies on the integrity of their code and data. A number of existing run-time defenses, like CPI and CFI, require to keep their data within the process of the vulnerable application to avoid a high performance overhead. Thus, the attacker may leverage a memory-corruption vulnerability in the application to bypass those defenses [21]. Traditionally, defense developers enforce the integrity of the (static) code using W⊕X or execute-only memory, while the integrity of the data relies on some form in-process memory isolation. However, existing memory isolation techniques, namely instrumentation and data hiding, force the defense developers to choose between high performance overheads and compromised security. IMIX ③ provides an efficient, secure, hardware-enforced in-process memory isolation mechanism. Data belonging to run-time mitiga-

tions is allocated in *isolated pages*, which are marked with a special access permission. We introduce a new dedicated instruction, `smov` ④, to access this data, while normal code belonging to the potentially vulnerable application is denied access to the isolated pages.

In addition to the `smov` instruction and the associated access permissions, IMIX includes a kernel extension ⑤ and compiler support ⑥. The kernel extension enables protected memory allocation by supporting the special access permission. IMIX's compiler integration enables applications as well as run-time defenses to leverage our memory isolation through high-level and low-level constructs for protected memory allocation and access. This makes it easy to adopt IMIX without detailed knowledge of IMIX's implementation.

In the following, we explain the individual building blocks of our IMIX framework in detail.

Hardware. For IMIX, we extend two of the CPU's main responsibilities, instruction processing and memory management. We add our `smov` instruction to the instruction set, reusing the logic of regular memory access instructions, so that the `smov` instruction has the same operand types of regular memory-accessing `mov` instructions, `mov` instructions without a memory operand do not need to be handled. The memory access logic is modified so that it will generate a fault if 1) an instruction other than `smov` is used to access a page protected by IMIX, or if 2) an `smov` instruction is used to access a normal page. Access by normal instructions to normal memory, and by `smov` instructions to protected memory, are permitted. If we allowed `smov` to access normal memory, attacks on metadata would be possible, e.g., the attacker could overwrite a pointer to CPI's metadata with an address pointing to an attacker-controlled buffer in normal memory. Our design ensures instructions intended to operate on secure data cannot receive insecure input.

Kernel. An operating system kernel controls the user-space execution environment and hardware devices. The kernel manages virtual memory using *page tables* that map the address of each page to the physical page frame that contains it. Each page is described by a *page table entry*, which also contains some metadata, including the access permissions for that page. A user-space program can request a change in its access permissions to a page through a system call.

We extend the kernel to support an additional access permission, which identifies all pages protected by IMIX. This enables protected memory allocation not only for statically compiled binaries, but also for code generated at run time, which has been an attractive target for recent attacks [23].

Compiler. A compiler makes platform functionality available as high-level constructs to developers. Its main objective is to transform source code to executables for a particular platform. We extend the compiler on both ends. First, IMIX provides two high-level primitives: one for allocating protected memory and one for accessing it. These memory-protection primitives can either be used to build mitigations, or to protect sensitive data directly. IMIX provides optimized interfaces for both use cases. Mitigations like CPI are implemented as an LLVM optimization pass that works at the intermediate representation (IR) level. IMIX provides IR primitives to use for IR modification. For application developers, IMIX provides source code annotations: variables with our annotation will be allocated in protected memory, and all accesses will be through the `smov` instruction.

5 Implementation

Figure 2 provides an overview of the components of IMIX. Developers can build programs with IMIX, using our extended Clang compiler ①, which supports annotations for variables that should be allocated in protected memory and new IR instructions to access the protected memory. We also modified its back end to support `smov` instructions. Programs protected by IMIX mark isolated pages using the system call `mprotect` with a special flag ②. Therefore, we extended the kernel's existing page-level memory protection functionality to support this flag and mark isolated pages appropriately ④. User-space programs access normal memory using regular instructions, e.g., `mov`, while accesses to protected memory must be performed using the instruction `smov` ③. To support IMIX, the CPU must be modified to support the `smov` instruction ⑤ and must perform the appropriate checks when accessing memory ⑥. In the following we explain each component in detail.

5.1 CPU Extension

As we mentioned in Section 4, every isolated page needs to be marked with a special flag. The CPU already has a data structure to store information about every page, which is called a Page Table Entry (PTE). In addition to the physical address of every virtual page, a PTE stores other metadata about the page, including permissions like writable and executable. Those flags are checked by the Memory Management Unit (MMU) to prevent unintended accesses. To implement our proof of concept, we mapped the IMIX protection flag to an ignored bit in the PTE; specifically, we chose bit 52, as it is the first bit not reserved, and is normally ignored by the MMU [31].

To enforce hardware protection, the CPU needs to be updated to enforce our access policy: non-`smov` can only

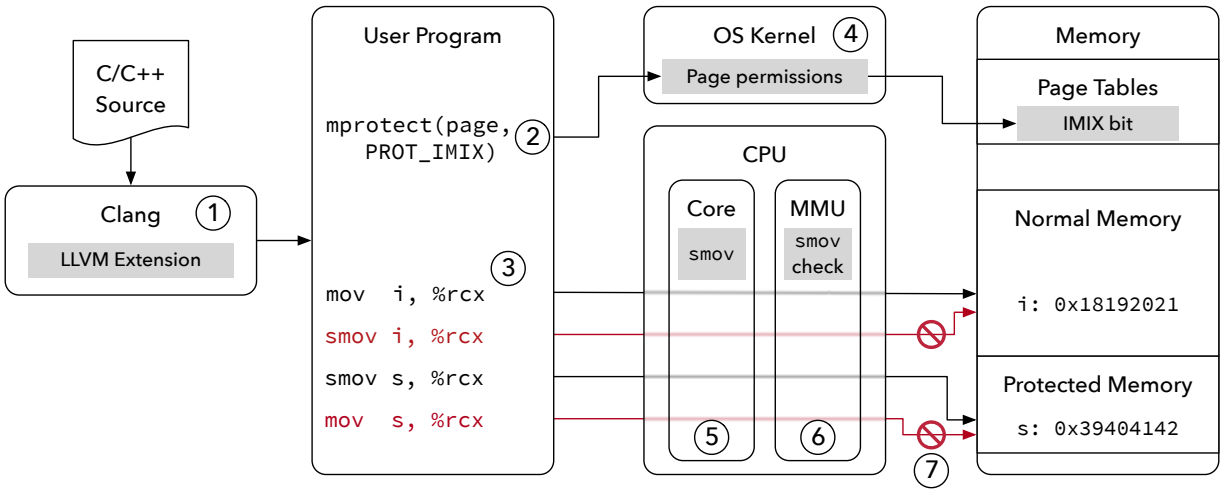


Figure 2: Overview of IMIX.

access regular pages, while `smov` can only access isolated pages. In other cases, the CPU must generate a fault (⑦ in Figure 2). The implementation of this logic requires the modification of the x86-64 ISA, which is challenging without source code access. Thus, we used a hardware simulator to show the feasibility of our design. Next, we describe how we extend x86-64 with the help of Intel's SAE, and then discuss the necessary modification to real hardware.

Simulated Hardware. We use Wind River Simics [64], a full system simulator, in order to simulate a complete computer which supports IMIX. Yet, Simics alone is too slow to boot the Linux kernel and test our kernel extension. Therefore we use the complementary Intel Simulation and Analysis Engine (SAE) add-on by Chachmon et al. [11]. Below we will refer to the system composed by Simics and SAE as simply SAE. SAE supports emulating an x86 system running a full operating system with its processes, while allowing various architectural instrumentations, including the CPU, the memory, and related hardware such as the memory management unit (MMU). This is done using extensions, called *ztools*, that may be loaded and unloaded at any time during emulation. They are implemented as shared libraries written in C/C++.

To instrument a simulated system, ztools registers callbacks for specific hooks either at initialization time or dynamically. First, we make sure that our ztool is initialized by registering a callback for the initialization hook. Then, we register a callback that is executed when an instruction is added to the CPU's instruction cache. If either a `mov` or `smov` instruction that accesses memory is found, we register an instruction replacement callback. Our registered callback handler can replace the instruction (using a provided C function), or execute the original instruction. In this handler, we implement IMIX's access logic. First, we check the protection flag of the memory accessed by the instruction. To identify protected memory, we look up the related PTE by combining the virtual address and the base address of the page table hierarchy linked from the `CR3` register. Our ztool then checks the IMIX page flag we introduced in the PTE.

If a regular instruction attempts to access regular memory, we execute the original instruction to avoid instruction cache changes. For `smov` instructions attempting to access an isolated page, we first remove the instruction from the instruction cache, and then execute our ztool implementation of this instruction. In the remaining cases, namely `smov` attempting to access regular memory, and regular instructions attempting to access isolated pages, we raise a fault.

Real Hardware. Adding IMIX support to a real CPU would require extending the CPU's instruction decoder to make it aware of our `smov` instruction. `smov` requires the same logic as the regular `mov` instruction, so the existing implementation could be reused. Moreover, we need to modify the MMU to perform the necessary checks. Analogously to W⊕X, we check the flag in the page table entry (PTE) belonging to the virtual address, and either permit or deny memory access. Modern MMUs are divided into three major components: logic for memory protection and segmentation, the translation lookaside buffer (TLB) which caches virtual to physical address mappings, and page-walk logic in case of a cache miss [49]. Our extension only modifies the first component to implement the access policy based on the current CPU instruction. Other components do not need to be modified, as we are using an otherwise ignored bit in the

PTEs. In Section 8 we discuss the feasibility of our proposed modification.

5.2 Operating System Extension

Access restrictions to the isolated pages are enforced by the hardware, without any involvement from the kernel. However, the isolated pages need to be marked as such in the PTEs, which are located in kernel memory. To support this, we modified a recent version of the Linux kernel. Specifically, we modified the default kernel for the Ubuntu 16.04 LTS distribution which is 4.10 at the time of writing. Similarly to W⊕X, we use page permissions to represent this information. Processes can request the kernel to mark a page as an isolated page by using the existing `mprotect` system call, which is already used to manage the existing memory access permissions: `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`. For IMIX, we add a dedicated `PROT_IMIX` boolean flag. The implementation of `mprotect` sets permission bits in the PTE according to the supplied protection modes. Note that once a page is marked as `PROT_IMIX` the only way to remove this flag from a page is by unmapping it first which will also set the memory to zero.

5.3 Compiler Extension

To provide C/C++ support for IMIX, we modify the LLVM compiler framework [40]. We chose LLVM over GCC because the majority of memory-corruption defenses leverage LLVM [16, 57, 66]. We modified the most recent version of LLVM (version 5.0) and ported our changes to LLVM 3.3 which is used by CPI [38].
Our modification mainly concerns the intermediate representation (IR) to provide access to the `smov` instruction to mitigations like CPI [38], and the x86 backend to emit the instruction. Further, we introduced an attribute that can be used to protect a single variable by allocating it in an isolated page, e.g., to protect a cryptographic secret. Next, we explain each modification in detail.

IR Extension. Run-time defenses are usually implemented as LLVM optimization passes that interact with and modify LLVM's intermediate representation. In order to allow those defenses to generate `smov` instructions, we extended the IR instructions set. The IR provides two memory accessors, specifically *load* and *store*, which represent respectively a load instruction from the memory to a temporary register, and a store instruction from a temporary register to the memory. Hence, we created two corresponding IMIX instructions: *sload* and *sstore*, which defense developers can use as a drop-in replacement for their regular counterparts.

LLVM IR instructions are implemented as C++ classes and therefore supports inheritance. We implemented our IR instructions to as subclasses of their regular counterparts in order to reuse the existing translation functionality from LLVM IR to machine code, called *lowering* in LLVM parlance.
To allocate memory in the isolated pages, we implemented an LLVM function that can be called from an optimization pass, which allocates memory at page granularity using `malloc` and immediately sets the IMIX permission using `mprotect`. A reference to the allocated memory is returned so that IMIX IR instructions can access the protected memory.

Attribute Support. Data-only attacks are hard to mitigate in practice. To give developers an efficient way to protect sensitive data like cryptographic keys at source code level, we added a IMIX attribute which can be used to annotate C/C++ variables which should be allocated in isolated pages. All instructions accessing those annotated variables will use the IMIX IR instructions instead of the regular ones. LLVM's `annotate` attribute allows arbitrary annotations to be defined, so we only needed to provide the logic needed to process our attribute. We implemented this as an LLVM optimization pass that replaces regular variable allocations with indexed slots in a IMIX protected safe region (one per compilation module), and changes all accessors accordingly.

Modifications to x86 Back End. In the back end, we added code needed to process *sload* and *sstore* instructions. In LLVM, the process of lowering IR instructions to machine code is two-staged. First, the *FastEmit* mechanism is used. It consists of transformation rules explicitly coded in C++ that are too complex to be processed using regular expressions. These are mainly platform-specific optimizations and workarounds. The mechanism can be used to either generate machine code directly, or to assign a rule that should be applied in the next stage. In the second stage, LLVM applies rule-based lowering using pattern matching. The IR instruction and its operands are matched against string patterns in LLVM's TableGen definitions, which define rules to lower the IR to the platform-specific machine code. We modified both stages of the lowering process, similarly to how *load* and *store* are handled.

5.4 Case Study: CPI

To evaluate the impact of our lightweight memory isolation technique to the performance, we ported Code-Pointer Integrity (CPI) by Kuznetsov et al. [38] to use IMIX. CPI uses a safe region in memory to guarantee integrity of code pointers and prevent code-reuse attacks.

All code pointers, pointers to pointers, and so on, are moved to the safe region, so that memory corruption vulnerabilities cannot be exploited to overwrite them. Return addresses are protected using a shadow stack. In contrast to its x86-32 implementation that leverages segmentation, CPI relies on hiding for x86-64 to protect the safe region. CPI places the safe region at a random address and stores this address in a segment, which is selected using the segment register `%gs`. During compilation, CPI's optimization pass moves every code pointer and additional metadata about bounds to the safe region. In order to access the safe region, CPI provides accessors that use `mov` instructions with a `%gs` segment override, which access the safe region using `%gs` as the base address and an offset. These accessors are provided by a compiler runtime extension which is linked late in compilation process. Evans et al. show that this CPI implementation is vulnerable, since the location of the safe region can be brute-forced [22].

We replaced data hiding with IMIX as the memory isolation technique used to prevent unintended accesses to CPI's safe region (including the shadow stack). First, we changed CPI's memory allocation function to not only allocate the safe region, but also set the IMIX protection flag. Second, we modified the compiler runtime, which provides access to the safe region, to make use of our `smov` instruction. Specifically, we changed the safe region functions to access memory directly via `smov` instructions instead of using register-offset addressing. This increases security of CPI dramatically. Since IMIX provides deterministic protection of the safe region, we do not need to prevent spilling of the safe region base address (stored in `%gs`), which IMIX makes CPI leakage resilient. Thus, knowing or brute-forcing the memory location brings no benefit any more, and prevents attacks like "Missing the Point(er)" by Evans et al. [22].

6 Security Analysis

The main objective of IMIX is to provide in-process memory isolation for data in order to make it accessible only by trusted code. Hence, the goal of an attacker is to access the isolated data. As IMIX is a hardware extension, an attacker cannot directly bypass it, i.e., use a regular memory access instruction to access the isolated memory. Thus, the attacker relies on creating or reusing *trusted code*, or manipulating the *data flow* to pass malicious values to the trusted code, or access to the configuration interface of IMIX.

Attacks on Trusted Code. As mentioned in our adversary model, IMIX assumes mitigations preventing the attacker from injecting new code [3], or reusing existing code [7, 50, 52, 54]. This prevents attackers from injecting `smov` instructions that are able to access the isolated data, or reusing trusted code with unchecked arguments, or exploiting unaligned instructions. This assumption is fulfilled by existing mitigations: the strict enforcement of W⊕X [44, 48] prevents the attacker from marking data as code, or changing existing code. Mitigations, such as Control-flow Integrity (CFI) [1, 45, 59] and Code-Pointer Integrity (CPI) [38] prevent the attacker from reusing trusted code.

Attacks on Data Flow. In general, attacks on the data flow [12, 19, 23, 28, 29] are hard to prevent since it would require the ability to distinguish between benign and malicious input data, which generally depends on the context. Therefore, the trusted code must either ensure that its input data originates from isolated pages protected by IMIX, or sanitize the data before using it. The former can be ensured by using the `smov` instruction to access the input data as IMIX's design ensures that the `smov` instruction cannot access unprotected memory. The latter heavily depends on the ability of the defense developer to correctly block inputs that would allow the attacker to manipulate the data within the protected memory in a malicious way: IMIX merely provides a primitive to isolate security critical data. Hence, if the developer fails to sanitize the input data in the trusted code, the code is vulnerable to data-flow attacks independently of whether it leverages IMIX or not. In practice, however, sanitizing inputs correctly requires limited complexity, e.g., in the case of a shadow stack [18] or CPI's safe region [38].

Attacks on Configuration. A common way to *bypass* mitigations is to disable them. For example, to bypass W⊕X, real-world exploits leverage code-reuse attacks to invoke a system call to mark a data buffer as code before executing it.

There are two ways for an attacker to re-configure IMIX: 1) leveraging the interface of the operating system to change memory permissions, or 2) manipulating page table entries.

For the first case, we assume that the attacker is able to manipulate the arguments of a benign system call to change memory permissions (`mprotect()` on Linux). Our design of IMIX's operating system support prevents the attacker from re-mapping protected memory to unprotected memory. Further, before IMIX memory is unmapped, the kernel sets the memory to zero to avoid any form of information disclosure attacks. Similarly, the kernel initializes memory, which is re-mapped as IMIX memory, with zeros to prevent the attacker from initializing memory with malicious values, mapping it as IMIX

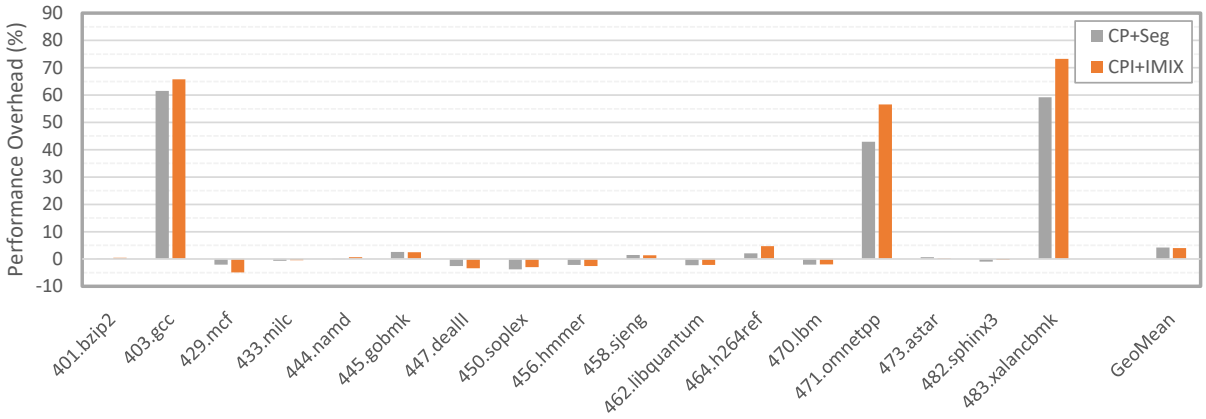


Figure 3: Performance overhead of CPI with segmentation-based memory hiding, and with IMIX.

memory, and then passing it to trusted code. Therefore, the developer must be aware that the attacker is potentially able to pass a pointer into a *zero-filled* page as an input value to trusted code.

For the second case, we assume that the attacker is able to exploit a memory-corruption vulnerability in the kernel. While the focus of this paper is on how user-mode defenses can leverage IMIX, our design allows kernel-based defenses to leverage IMIX as well. Hence, to mitigate data-only attacks against the page table [19] existing defenses [17, 25] can leverage IMIX to ensure that only trusted code can access the page tables.

7 Performance Evaluation

To evaluate the performance of our approach, we ported the original implementation of CPI by Kuznetsov et al. [38] to leverage IMIX to isolate the safe region and applied it to the SPEC CPU2006 benchmark suite. Specifically, we executed all C/C++ benchmarks with the reference workload to measure the performance overheads. The SPEC CPU2006 benchmarking suite is comprised of CPU-intensive benchmarks that frequently access memory, and hence, are well suited to evaluate our instrumentation. We performed our evaluation using Ubuntu 14.04 LTS with Linux Kernel version 3.19.0 on an Intel Core i7-6700 CPU in 64-bit mode running at 3.40 GHz with dynamic voltage and frequency scaling disabled, and 32 GB RAM.

Baseline. First, we measured the performance impact of the original CPI implementation, which we obtained from the project website [39]. Despite efforts, we were unable to execute the CPI-instrumented version of *perlbench* and *povray*. Using the geometric mean of positive overheads, we measured a performance overhead of 4.24% (arithmetic mean of 9.05%, Kuznetsov et al. [38] measured an average performance overhead of 8.4%). We measured a maximum overhead of 61.49% (*gcc*), while a maximum of 44.2% (for *omnetpp*) was reported in the original paper.

CPI with IMIX. Next, we evaluated the performance overhead of IMIX. As hardware emulation turned out to be too slow for executing the SPEC CPU2006 benchmarking tests, we instead evaluated IMIX by replacing `smov` instructions with `mov` instructions that access memory directly. We argue that this reflects the actual costs for `smov` instructions, because the IMIX permission check is part of the paging permission check.

During our performance evaluation we made the interesting observation that our IMIX instrumentation comes with a higher overhead than the baseline. In total, we measured a performance overhead of 14.70% for IMIX, which is an increase of 1.94% in comparison to segmentation-based CPI. In addition, we observed a maximum overhead of 73.27%, compared to a maximum of 61.49% for segmentation-based CPI.

We further investigated this counter-intuitive result. First, we verified with the help of a custom micro-benchmarks that the access time to a memory buffer through a segment register is consistently faster than just dereferencing a general purpose register. Interestingly, it makes no difference whether the base address of the segment is set to 0 or the base address of the buffer. Second, we found that the faster access through segment registers is, at least partially, related to the L2 hardware prefetcher: when we disable it, memory accesses through a general purpose register are faster than segment-based accesses (difference in geometric mean is 0.47% in SPEC CPU2006).

Technique	Policy-based Isolation	Hardware Enforced	Fast Interleaved Access	Fails Safe
SFI	✓	×	✓	×
Segmentation	only for x86-32	✓	✓	✓
Memory Hiding	×	×	✓	×
Paging / EPT	only single-threaded applications	✓	×	✓
Intel MPK	✓	✓	✓	✓
Intel SGX	✓	✓	×	✓
Intel MPX	✓	✓	✓	×
Intel CET	only for Shadow Stack	✓	✓	✓
SMOV	✓	✓	✓	✓

Table 1: Comparison of memory-isolation techniques. Legend: *Policy-based Isolation* means that the memory protection itself cannot be bypassed with an arbitrary memory read-write primitive. *Hardware Enforced* is self-explanatory. *Fast Interleaved Access* refers to the ability to alternately access protected and unprotected memory without additional performance impact. *Fails Safe* means that regular (un-instrumented) memory instructions cannot access the protected memory.

CPI with IMIX (Segment-based Addressing). Similarly to a regular `mov` instruction, the IMIX instruction allows to access memory through a segment register. Unsurprisingly, by adjusting our IMIX-based CPI instrumentation to use segment register-based addressing we achieve 0% overhead over CPI. We further compare IMIX to other memory protection approaches, namely Intel MPK and Intel MPX, in Section 9.

8 Discussion

On the Feasibility of Our ISA Extension. One of the main values of any defense in the field of system security is practicality. Therefore, it comes with no surprise that existing research often sacrifices security in favor of performance [45, 53, 67], and retrofit existing hardware features [6, 16, 18, 41, 58, 63] instead of introducing more suitable ones. The reason is that in practice it is unlikely that hardware vendors are going to change their hardware design and risk compatibility issues with legacy software in order to strengthen the security and increase the performance of a specific mitigation.

However, we argue that this does not apply to IMIX for two reasons: 1) IMIX enables strong and efficient in-process isolation of data which is an inevitable requirement of many memory-corruption defenses. 2) IMIX can be implemented by slightly modifying Intel's proposal, Control-flow Enforcement Technology (*CET*) [33].

As we discussed in Section 2, memory-corruption defenses often reduce the attack surface from potentially the whole application's memory to the memory that is used by the defense itself. With IMIX we provide a strong and efficient hardware primitive to enforce the protection of this data which is mitigation-agnostic. By providing a primitive, which is essential to memory-corruption defenses, rather than implementing a specific defense in hardware [33], vendors avoid the risk of a later bypass [50].

We believe that IMIX can be adopted in real world with comparatively low additional effort. With CET [33] Intel provides a specialization of IMIX. Similar to IMIX, CET requires modifications to the TLB, semantic changes to the page table, and the introduction of new instructions. Contrary to IMIX, CET's hardware extension is tailored to isolate the shadow stack of a CFI implementation [45]. As expected, generalizing CET's shadow stack to support arbitrary memory accesses still allows implementation of an isolated shadow stack [18].

9 Related Work

In the following, we discuss techniques that may be used to protect memory against unintended access. Table 1 provides an overview of characteristics of these techniques. We explain each of its aspects in detail, and compare them to IMIX.

Software-based Memory Protection. Software-fault isolation techniques (SFI) [51, 61] allow to create a separate protected memory region. SFI is implemented by instrumenting every memory-access instruction such that the address is masked before the respective instruction is executed. This ensures that the instrumented instruction can only access the designated memory segment, however, this instrumentation also has a significant performance impact. Though SFI instruments every load/store

instruction, invalid memory accesses cannot be detected, but are instead masked to point to unprotected memory [37]. ISboxing [20] leverages instruction prefixes of x86-64 to implicitly mask load and store operations. The instruction prefix determines whether a memory-access instruction uses a 32-bit (default case) or 64-bit address. By ensuring that untrusted code can only use 32-bit addresses to access memory, protected data can be stored in memory that can only be addressed with 64-bit addresses. Yet, this reduces the available address space significantly, and allows linked libraries to access protected memory.

Another way of protecting data against malicious modifications is to enforce data-flow integrity (DFI) [2, 10, 55]. DFI creates a data-flow graph by means of static analysis, which is enforced during run time by instrumenting memory-access instructions. However, the performance overhead of DFI, which e.g. is on average 7% for WIT [2], prevents it from being used to safeguard protection secrets of code-reuse mitigations, since it would further increase the mitigation's performance overhead. IMIX can be used for both protecting sensitive data (like DFI does) and enabling efficient protection of safe regions for control-flow hijacking mitigations.

Retrofitting Existing Memory Protection. Segmentation is a legacy memory-isolation feature on x86-32 that allows to split the memory into isolated segments [61, 65]. For memory accesses, the current privilege level is checked against the segment's required privilege level directly in hardware. On x86-64 segmentation registers still exist but access control is no longer enforced [37]. On the surface, re-enforcing legacy segmentation seems to be an attractive solution, however, IMIX is easier to implement from a hardware perspective: segmentation requires arithmetic operations, IMIX only one check. Moreover, IMIX provides higher flexibility: protected memory does not need to consist of one contiguous memory region. As segmentation registers are rarely used by regular applications any more, they are often used to store base addresses for memory hiding [6, 38, 41]. Indeed, segmentation-based memory hiding comes with no performance overhead, however, unlike IMIX, it does not provide real in-process isolation and is vulnerable to memory-disclosure attacks [22, 26].

Paging can also be used as well to provide in-process isolation by removing read/write permissions from a page when executing untrusted code [5]. However, regularly switching between trusted and untrusted code is expensive because of 1) two added `mprotect()` system calls, and 2) the following invalidation of TLB entries for each of them [60]. Further, this technique is vulnerable to race-condition attacks, i.e., the attacker can access the protected data from a second thread that runs concurrently to the trusted code. IMIX avoids both disadvantages.

A more recent feature introduced with Intel VT-x is Extended Page Tables (EPT) [32] to implement hardware-assisted memory virtualization. EPT provide another layer of indirection for memory accesses that is controlled by the hypervisor but is otherwise conceptually the same as regular paging. Additionally, VT-x introduces an instruction, `vmfunc`, that enables fast switches between EPT mappings. Hence, to isolate memory, the hypervisor maintains two EPT mappings [16] (regular and protected memory) and trusted code invokes the `vmfunc` instruction instead of `mprotect()`). However, this approach suffers from the same disadvantages as the previous approach which relies on regular paging.

Proposed Memory Isolation Mechanisms. There are already several academic proposals for memory isolation. HDFI [56] is a fine-grained data isolation mechanism that uses MMU tagging for RISC-V. However, due to the need of an additional tag table, HDFI needs two accesses per memory operation. Thus, HDFI leverages additional hardware units (like a cache) to lower the performance impact. Still, HDFI relies on complex static analysis for data-flow integrity which does not meet the requirements for modern JIT-compiled code. IMIX supports JIT compilation by building on existing functionality like `mprotect`, furthermore, IMIX does not need any additional static analysis.

CHERI [62] extends a RISC architecture with fine-grained memory isolation using a set of ISA extensions. For this, two compartments are introduced, however, switching costs are comparably high (620 cycles overhead). In addition, CHERI also relies on intensive static analysis unsuitable for JIT code.

ILDI [13] is another data isolation approach, but for ARM. It leverages existing ARM features (`Privileged Access Never`, `PAN`) to create a safe region for sensitive kernel memory, isolated from potential kernel exploits. By explicitly granting `Load and Store Unprivileged` (`LSU`) instructions access to sensitive data, regular accesses (possibly attacker controlled) are no longer allowed to access the safe region. However, ILDI imposes a high performance overhead on the kernel (35.3%). IMIX proposes a general approach that can be leveraged by both kernel-space and user-space mitigations.

Recent Hardware Extensions. Recent Intel CPUs implement a variety of new memory-protection features. In particular, Memory Protection Extensions (MPX) and Memory Protection Keys (MPK) can be retrofitted to enable in-process memory isolation. Nevertheless, as we discuss in the following, they are not viable alternatives

Name	CPI+Seg (%)	CPI+IMIX (%)	CPI+MPK (%)	CPI+MPX (%)
400.perlbench	-	-	-	-
401.bzip2	0.13	0.44	0.19	132.36
403.gcc	61.49	65.73	2856.48	-
429.mcf	-2.08	-4.89	-2.41	203.71
433.milc	-0.63	-0.47	-0.45	-6.36
444.namd	-0.10	0.66	-0.09	-8.60
445.gobmk	2.55	2.52	32.41	-
447.dealII	-2.57	-3.37	-	-
450.soplex	-3.83	-2.96	-0.74	2.88
453.povray	-	-	-	-
456.hmmer	-2.17	-2.54	-1.35	15.43
458.sjeng	1.43	1.36	1.39	56.81
462.libquantum	-2.32	-2.16	-2.62	106.41
464.h264ref	2.04	4.67	536.02	46.87
470.lbm	-2.04	-1.99	-1.94	-9.82
471.omnetpp	42.95	56.62	1444.02	-
473.astar	0.67	0.20	0.70	-1.29
482.sphinx3	-0.99	-0.32	5.52	-0.68
483.xalancbmk	59.23	73.27	1385.67	-
GeoMean	4.24	3.99	12.43	36.86

Table 2: Comparison of memory isolation techniques. *CPI+Seg* uses memory hiding to protect the safe region, for the remaining the respective technique is used. Note that entries marked with "-" crashed with CPI applied.

to IMIX as both come with disadvantages that render them impractical.

The main goal of MPX [31] is to provide hardware-assisted bounds checking to avoid buffer overflows. Therefore, the developer specifies bounds using dedicated registers (each contains a lower and an upper bound) that can be checked by newly introduced instructions. MPX can be retrofitted to enforce memory isolation by defining one bound that divides the address space in two segments: a regular, and a protected region. Then, bounds checks are inserted for every memory access instruction that is not allowed to access protected memory [37]. This has two main disadvantages. First, MPX does not fail safe, i.e., not instrumented instructions (by a third-party library, for example) can still access the safe region. Second, instructions that are allowed to access protected memory can still access unprotected memory. Hence, an attacker might be able to redirect memory accesses of trusted code to attacker-controlled memory. To avoid such attacks, additional instrumentation of the trusted code is required, which significantly increases the performance overhead, as depicted in Table 2. Protecting CPI's safe region with MPX using the open-source implementation by Koning et al. [37] results in a total performance overhead of 36.86% with a maximum of 203.71% for *mcf*, which cannot be considered practical, especially since we were not able to execute the benchmarks that show the highest overheads across all techniques. In comparison, IMIX is secure by default, and enforces strict isolation between protected and unprotected memory without additional overhead.

Intel's MPK is a feature to be available in upcoming Intel x86-64 processors [27, 34], already available on other architectures like IA-64 [30], and ARM32 (called *memory domains*) [4]. Since IMIX and MPK implement a similar idea, we also evaluated MPK based on the approximation given by Koning et al. [37] using the setup we describe in Section 7.

As shown in Table 2, using MPK to protect the CPI safe region results in a total performance overhead of 12.43% with a maximum of 2856.48% for *gcc*. We identified the additional instrumentation to switch between trusted and untrusted code to be the root cause of the additional overhead. This emphasizes the conceptual differences of MPK and IMIX. MPK enables many distinct domains to be present. Reducing these to two possible domains allows IMIX to be leveraged by mitigations like CPI or CFI that rely on frequent domain switches. In contrast, MPK is useful if the application changes domains infrequently, i.e., for temporal memory isolation, or to isolate different threads.

Encryption can also be used to protect memory. For instance, Intel Total Memory Encryption [35] (Secure Memory Encryption for AMD [36]) allows to encrypt the whole memory transparently, protecting it from physical analysis like cold-boot attacks, but not local memory corruption attacks [37]. Another encryption feature, AES-NI [35], reduces overhead associated with encryption

dramatically, which can be used to encrypt and decrypt safe regions as needed. Even with hardware encryption support, solutions like CCFI still induce a performance overhead of up to 52% [42], and keeping the encryption key safe requires relying on unused registers and ensuring that this key is never spilled to memory [14, 37]. IMIX is not prone to register spilling, since it does not rely on a secret to protect memory.

Trusted Execution Environments like Intel SGX [15] offer strong security guarantees through hardware support, but require intensive effort to decouple code to be run in the enclave. SGX can also be used for memory protection, but only at high performance costs due to overheads for entering and exiting the enclave.

10 Conclusion

Mitigations against memory-corruption attacks for modern x86-based computer systems rely on in-process protection of their code and data. Unfortunately, neither current nor planned memory-isolation features of the x86 architecture meet these requirements. As a consequence, many mitigations rely on information hiding via segmentation, on expensive software-based isolation, or on retrofitting memory-isolation features that require compromises in the design of the mitigation.

With IMIX we design a mitigation-agnostic in-process memory-isolation feature for data that targets the x86 architecture. It provides memory-corruption defenses with a well-suited isolation primitive to protect their data. IMIX extends the x86 ISA with an additional memory permission that can be configured through the page table, and a new instruction that can only access memory pages which are isolated through IMIX. We implement a fully-fledged proof of concept of IMIX that leverages Intel's Simulation and Analysis Engine to extend the x86 ISA, and we extend the Linux kernel and the LLVM compiler framework to provide interfaces to IMIX. Further, we enhance Code-pointer Integrity (CPI), an effective defense against code-reuse attacks, using IMIX to protect CPI's safe region.

Our evaluation shows that defenses, like CPI, greatly benefit from IMIX in terms of security without additional performance overhead. We argue that the adoption of IMIX is possible by adjusting the design of Intel's Control-flow Enforcement Technology (CET). Finally, IMIX provides a solution that can serve as a building block for forthcoming defenses to tackle challenging problems, such as data-oriented attacks.

Acknowledgments. This work was supported by the German Science Foundation CRC 1119 CROSSING P3, the German Federal Ministry of Education and Research (BMBF) in the context of HWSec, and the Intel Collaborative Research Institute for Collaborative Autonomous and Resilient Systems (ICRI-CARS).

11 Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *29th IEEE Symposium on Security and Privacy*, S&P, 2008.

[3] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49, 2000.

[4] ARM. ARM architecture reference manual. `http://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet9/DDI0487A_h_armv8_arm.pdf`, 2015.

[5] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2014.

[6] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[7] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[8] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium*, NDSS, 2016.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

[10] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2006.

[11] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi. Simulation and analysis engine for scale-out workloads. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 22:1–22:13, New York, NY, USA, 2016. ACM.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, USENIX Sec, 2005.

[13] Y. Cho, D. Kwon, and Y. Paek. Instruction-level data isolation for the kernel on arm. In *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pages 1–6. IEEE, 2017.

[14] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[15] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[16] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R.

Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[17] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *35th IEEE Symposium on Security and Privacy*, S&P, 2014.

[18] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2015.

[19] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Pt-rand: Practical mitigation of data-only attacks against page tables. 2017.

[20] L. Deng, Q. Zeng, and Y. Liu. Isboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP International Information Security Conference*, pages 386–400. Springer, 2015.

[21] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 781–796. IEEE, 2015.

[22] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[23] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi. Jitguard: Hardening just-in-time compilers with sgx. In *24th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2017.

[24] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS*, 2016.

[25] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing kernel code integrity on the trustzone architecture. In *Mobile Security Technologies*, MoST, 2014.

[26] E. Göktaş, R. Gawlik, B. Kollenda, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *25th USENIX Security Symposium (USENIX Security 16)*, pages 105–119. USENIX Association, 2016.

[27] D. Hansen. [rfc] x86: Memory protection keys. `https://lwn.net/Articles/643617/`, 2015.

[28] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium*, USENIX Sec, 2015.

[29] H. Hu, S. Shinde, A. Sendroiu, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *37th IEEE Symposium on Security and Privacy*, S&P, 2016.

[30] Intel. Intel Itanium architecture developer's manual: Vol. 2. `https://www.intel.de/content/dam/www/public/us/en/documents/manuals/itanium-architecture-software-developer-rev-2-3-vol-2-manual.pdf`, 2010.

[31] Intel. Intel 64 and IA-32 architectures software developer's manual, combined volumes 3A, 3B, and 3C: System programming guide. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf`, 2013.

[32] Intel. Intel 64 and IA-32 architectures software developer's manual. ch 28, 2015.

[33] Intel. Control-flow Enforcement Technology Preview, 2017.

[34] Intel. Intel 64 and IA-32 architectures software developer's manual. `http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2017.

[35] Intel. Intel architecture memory encryption technologies specification. `https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf`, 2017.

[36] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. *White paper*, 2016.

[37] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452. ACM, 2017.

[38] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[39] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. CPI implementation. `http://dslab.epfl.ch/proj/cpi/levee-early-preview-0.2.tgz`, 2014.

[40] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2004.

[41] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[42] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: cryptographically enforced control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[43] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium*, USENIX Sec, 2006.

[44] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[45] Microsoft. Control flow guard. `http://msdn.microsoft.com/en-us/library/Dn919635.aspx`, 2015.

[46] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2009.

[47] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In *International Symposium on Memory Management*, ISMM, 2010.

[48] OpenBSD. Openbsd 3.3, 2003.

[49] J. Power, M. D. Hill, and D. A. Wood. Supporting x86-64 address translation for 100s of gpu lanes. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 568–578. IEEE, 2014.

[50] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications.

In *36th IEEE Symposium on Security and Privacy*, S&P, 2015.

[51] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *18th USENIX Security Symposium*, USENIX Sec, 2010.

[52] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2007.

[53] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2004.

[54] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[55] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[56] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: hardware-assisted data-flow isolation. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 1–17. IEEE, 2016.

[57] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[58] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[59] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium*, USENIX Sec, 2014.

[60] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 340–349. IEEE, 2011.

[61] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[62] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 20–37. IEEE, 2015.

[63] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *11th ACM Symposium on Information, Computer and Communications Security*, ASIACCS, 2016.

[64] Wind River. Simics full system simulator. `https://www.windriver.com/products/simics/`, 2018.

[65] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[66] C. Zhang, W. Zou, T. Wang, Y. Chen, and T. Wei. Using type analysis in compiler to mitigate integer-overflow-to-buffer- overflow threat. *Journal of Computer Security*, 19:1083–1107, 01 2011.

[67] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium*, USENIX Sec, 2013.

B

CURE: A Security Architecture with CUstomizable and Resilient Enclaves (USENIX Sec'21)

---

[13] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, A. Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021. CORE Rank A*. Chapter 3.

CURE: A Security Architecture with CUstomizable and Resilient Enclaves

Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf,
Technische Universität Darmstadt

https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

CURE: A Security Architecture with CUstomizable and Resilient Enclaves

Raad Bahmani, Ferdinand Brasser, Ghada Dessouky,
Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, Emmanuel Stapf
Technische Universität Darmstadt, Germany
{raad.bahmani, ferdinand.brasser, ghada.dessouky, patrick.jauernig,}
{matthias.klimmek, ahmad.sadeghi, emmanuel.stapf}@trust.tu-darmstadt.de

Abstract

Security architectures providing Trusted Execution Environments (TEEs) have been an appealing research subject for a wide range of computer systems, from low-end embedded devices to powerful cloud servers. The goal of these architectures is to protect sensitive services in isolated execution contexts, called *enclaves*. Unfortunately, existing TEE solutions suffer from significant design shortcomings. First, they follow a *one-size-fits-all* approach offering only a single enclave *type*, however, different services need flexible enclaves that can adjust to their demands. Second, they cannot efficiently support emerging applications (e.g., Machine Learning as a Service), which require secure channels to peripherals (e.g., accelerators), or the computational power of multiple cores. Third, their protection against cache side-channel attacks is either an afterthought or impractical, i.e., no fine-grained mapping between cache resources and individual enclaves is provided.

In this work, we propose CURE, the first security architecture, which tackles these design challenges by providing different types of enclaves: (i) *sub-space* enclaves provide vertical isolation at all execution privilege levels, (ii) *user-space* enclaves provide isolated execution to unprivileged applications, and (iii) *self-contained* enclaves allow isolated execution environments that span multiple privilege levels. Moreover, CURE enables the exclusive assignment of system resources, e.g., peripherals, CPU cores, or cache resources to single enclaves. CURE requires minimal hardware changes while significantly improving the state of the art of hardware-assisted security architectures. We implemented CURE on a RISC-V-based SoC and thoroughly evaluated our prototype in terms of hardware and performance overhead. CURE imposes a geometric mean performance overhead of 15.33% on standard benchmarks.

1 Introduction

For decades, software attacks on modern computer systems have been a persisting challenge leading to a continuous arms race between attacks and defenses. The ongoing discovery of exploitable bugs in the large code bases of commodity operating systems have proven them unsuitable for reliable protection of sensitive services [104, 105]. This motivated various hardware-assisted security architectures integrating *hardware security primitives* tightly into the System-on-Chip (SoC). Capability-based systems, such as CHERI [100], CODOMs [95], IMIX [30], or HDFI [82], offer fine-grained protection through (in-process) sandboxing, however, they cannot protect against privileged software adversaries (e.g., a malicious OS). In contrast, security architectures providing Trusted Execution Environments (TEE) enable isolated containers, also called *enclaves*. Enclaves allow for a coarse-grained but strong protection against adversaries in privileged software layers. TEE architectures have been proposed for a variety of computing platforms[1], in particular for modern high-performance computer systems, e.g., industry solutions like Intel SGX [35], AMD SEV [38], ARM TrustZone [3], or academic solutions such as Sanctum [22], Sanctuary [10], Keystone [48], or Komodo [27] to name some.

In this paper, we focus on TEE architectures for modern high-performance computer systems. We investigate the shortcomings of existing TEE architectures and propose an enhanced and significantly more flexible TEE architecture with a prototype implementation for the open RISC-V architecture.

Deficiencies of existing TEE architectures. So far, existing TEE architectures have adopted a *one-size-fits-all* enclave approach. They provide only one *type* of enclave requiring applications and services to be adapted to these enclaves' features and limitations, e.g., Intel SGX restricts system calls of its enclaves and thus, applications need to be modified when being ported to SGX which produces additional costs. Additional efforts like Microsoft's Haven framework [5] or Graphene [87] are needed to deploy unmodified applications to SGX enclaves. Moreover, today, we are using diverse

[1] TEE architectures for resource-constrained embedded systems (e.g., Sancus [66], TyTAN [8], TrustLite [47] or TIMBER-V [98]) are not the subject of this paper.

services that process sensitive data, e.g., payment, biometric authentication, smart contracts, speech processing, Machine Learning as a Service (MLaaS), and many more. Each service imposes a different set of requirements on the underlying TEE architecture. One important requirement concerns the ability to securely connect to devices. For example on mobile devices, privacy-sensitive data is constantly collected over various sensors, e.g., audio [9], video [83], or biometric data [19]. On cloud servers, massive amounts of sensitive data are aggregated and used to train proprietary machine learning models, often outside of the CPU, offloaded to hardware accelerators [84]. However, TEE architectures such as SGX [35], SEV [38] and Sanctum [22], do not consider secure I/O at all, solutions such as Keystone [48] would require additional hardware to support DMA-capable peripherals, solutions like Graviton [96] require hardware changes at the peripheral side. TrustZone [3], Sanctuary [10] and Komodo [27] cannot bind peripherals directly to individual enclaves.

Another important requirement imposed on TEE architectures is an adequate and practical protection against side-channel attacks, e.g., cache [11,50] or controlled side-channel attacks [65,92,101]. Current TEE architectures either do not include cache side-channel attacks in their threat model, like SGX [35], or TrustZone [3], only provide impractical solutions which heavily influence the OS, like Sanctum [22], or do not consider controlled side-channel attacks, e.g., SEV [38]. We will elaborate on the related work and the problems of existing TEE architectures in detail in Section 9.

This work. In this paper, we present a TEE architecture, coined CURE, that tackles the problems of existing solutions with a cost-effective and architecture-agnostic design. CURE offers multiple types of enclaves: (i) sub-space enclaves that isolate only parts of an execution context, (ii) user-space enclaves, which are tightly integrated into the operating system, and (iii) self-sustained enclaves, which can span multiple CPU-cores and privilege levels. Thus, CURE is the first TEE architecture offering a high degree of freedom in adjusting enclave boundaries to fulfill the individual functionality and security requirements of modern sensitive services such as MLaaS. CURE can bind peripherals, with and without DMA support, exclusively to individual enclaves. Further, it provides side-channel protection via flexible and fine-grained cache resource allocation.

Challenges. Building a TEE architecture with the described properties comes with a number of challenges. (i) New hardware security primitives must be developed that allow enclaves to adapt to different functionality and security requirements. (ii) Even though the security primitives should allow flexible enclaves, they must not require invasive hardware modification, which would impede cross-platform adoption. (iii) While the changes in hardware should remain small, performance overhead for managing enclaves in software must be minimized. (iv) Protections against the emerging threat of microarchitectural attacks in form of side-channel and transient-execution attacks must be considered in the design for all types of enclaves.

Contributions. Our design of CURE and its implementation on the RISC-V platform tackles all these challenges. To summarize, our main contributions are as follows:

- We present CURE, our novel architecture-agnostic design for a flexible TEE architecture which can protect unmodified sensitive services in multiple enclave types, ranging from enclaves in user space, over sub-space enclaves, to self-contained (multi-core) enclaves which include privileged software levels and support enclave-to-peripheral binding.
- We introduce novel hardware security primitives for the CPU cores, system bus and shared cache, requiring minimal and non-invasive hardware modifications.
- We prototype CURE for the open RISC-V platform using the open-source Rocket Chip generator [4].
- We evaluate CURE's hardware and software components in terms of added logic and lines of code, and CURE's performance overhead on an FPGA and cycle-accurate simulator setup using micro- and macrobenchmarks.

2 System Assumptions

CURE targets a modern high-performance multi-core system, with common performance optimizations like data and instruction caches, a Translation Lookaside Buffer (TLB), shared caches, branch predictors, respective instructions to flush the core-exclusive resources, and a central system bus that connects the CPU with the main memory (over a dedicated memory controller) and various peripherals.

System bus and peripherals. The system bus connects the CPU to a plethora of system peripherals over a fixed set of hardwired peripheral controllers. The peripherals range from storage, communication, and input devices to specialized compute units, e.g., hardware accelerators [37]. The CPU interacts with peripherals using parts of the internal peripheral memory which are mapped to the address space of the CPU, called Memory-Mapped I/O (MMIO). We assume that the CPU can nullify the internal memory of a peripheral to sanitize its state. Every access from the CPU to a peripheral is decoded in the system bus and delegated to the corresponding peripheral. The CPU acts as a *parent* on the system bus, whereas the peripherals (and main memory) act as *childs* that respond to requests from a parent. However, MMIO is not sufficient for some peripherals where large amounts of data need to be shared with the CPU since the CPU needs to copy the data from the main memory to the peripheral memory. Therefore, these peripherals are often connected to the system bus as *parents* over Direct Memory Access (DMA) controllers, allowing them to directly access the main memory. To cope with resource contention in these complex interconnects, system buses also incorporate arbitration mechanisms to schedule the

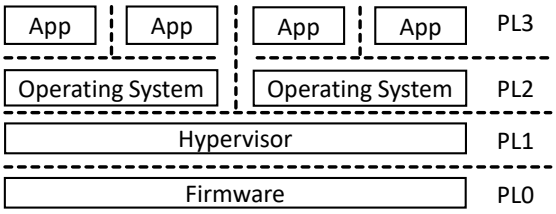


Figure 1: Software privilege levels (PL): user space, kernel space & dedicated levels for hypervisor & firmware.

establishment of parent-child connections when multiple bus requests occur simultaneously.

Software privilege levels. We assume the CPU supports the privilege levels (PLs) as shown in Figure 1. In line with modern processors (Intel [21], AMD [34] or ARM [55]), we assume a separation between a user-space layer (PL3) and a more privileged kernel-space layer (PL2), which is performed by the MMU (configured by PL2 software) through virtual address spaces. The CPU may support a distinct layer for hypervisor software (PL1) to run virtualized OS in Virtual Machines (VMs), where the separation to PL2 is performed by a second level of hardware-assisted address translation [73]. Lastly, we assume a highly-privileged layer (PL0) which contains firmware that performs specific tasks, e.g., hardware emulation or power management.

We assume that the system performs secure boot on reset, whereas the first bootloader stored in CPU Ready-Only Memory (ROM), verifies the firmware through a chain of trust [53]. After verification, the firmware starts execution from a predefined address in the firmware code and loads the current firmware state from non-volatile memory (NVM) where it is stored encrypted, integrity- and rollback-protected. The cryptographic keys to decrypt and verify the firmware state are passed by the bootloader which loads the firmware into Random-access Memory (RAM). Rollback protection can be achieved, e.g., by making use of non-volatile memory with Replay Protected Memory Block (RPMB) partitions or by using eFuses as secure monotonic counters [56]. When a system shutdown is performed, the firmware stores its state in the NVM, encrypted and integrity- and rollback-protected.

3 Adversary Model

Our adversary model adheres to the one commonly assumed for TEE architectures, i.e., a strong software-only adversary that can compromise all software components, including the OS, except a small software/microcode Trusted Computing Base (TCB) which configures the hardware security primitives of the system, manages the enclaves and which is inherently trusted [3, 10, 22, 27, 35, 48].

We assume that the goal of the adversary is to leak secret information from the TCB or from a victim enclave. An adversary with full control of the system software can inject own code into the kernel (PL2) and even into the hypervisor (PL1). This allows the adversary, with full access to the TCB interface used for setting up enclaves, to spawn malicious processes and even enclaves. Even though the adversary cannot change the firmware code (which uses secure boot), memory corruption vulnerabilities might still be present in the code and be exploitable by the adversary [24]. In addition, we assume that an adversary is able to compromise peripherals from software to perform DMA attacks [63, 76].

We assume the underlying hardware to be correct and trusted, and hence, exclude attacks that exploit hardware flaws [40, 86]. We also do not assume physical access, and thus, fault injection attacks [6], physical side-channel attacks [46, 62] or the physical connection of malicious peripherals are out of scope. We do not consider Denial-of-Service (DoS) attacks in which the adversary starves an enclave since an adversary with control over the OS can shut down the complete system trivially. As standard for TEE architectures, CURE does not protect from software-exploitable vulnerabilities in the enclave code but prevents their exploitation from compromising the complete system.

4 Requirements Analysis

To provide customizable, practical and strongly-isolated enclaves, CURE must fulfill a number of security and functionality requirements. We list them in the following section, and show in Section 7 how CURE fulfills the security requirements. In Section 6 and Section 8, we demonstrate how the functionality requirements are met.

4.1 Security Requirements (SR)

SR.1: Enclave protection. Enclave code must be integrity-protected when at rest, and inaccessible for an adversary when executed. All sensitive enclave data must remain confidential and integrity-protected at all times. An enclave must be protected from adversaries on all software layers (PL3-PL0), other potentially malicious enclaves, and DMA attacks [63, 76].

SR.2: Hardware security primitives. The protection of the enclaves must be enforced by secure hardware components which can only be configured by the software TCB.

SR.3: Minimal software TCB. The TCB must be protected from adversaries in all software layers (PL3-PL0) and minimal in size to be formally verifiable, i.e., a few KLOCs [44].

SR.4: Side-channel attack resilience. Mitigations against the most relevant software side-channel attacks must be available, namely, side-channel attacks on cache resources [31, 50, 70, 102], controlled side-channel attacks [65, 92, 101] and transient-execution attacks [12, 14, 43, 45, 78, 89, 90, 93].

4.2 Functionality Requirements (FR)

FR.1: Dynamic enclave boundaries. The trust boundaries of an enclave must be freely configurable such that enclaves

at different privilege levels can be supported.

FR.2: Enclave-to-peripheral binding. Secure communication between enclaves and selected system peripherals, e.g., when offloading sensitive machine learning tasks to hardware accelerators [84], must be explicitly supported.

FR.3: Minimal hardware changes. The hardware changes required to integrate the proposed security primitives into a commodity SoC (cf. Section 2) must be minimal, no invasive changes to CPU internals must be required to enable a higher adoption of CURE in future platforms.

FR.4: Reasonable performance overhead. The performance overhead incurred during enclave setup and run time must be minimized and must not render the computer system impractical for certain uses cases or degrade user experience.

FR.5: Configurable protection mechanisms. Protection mechanisms against cache side-channel attacks must be applicable dynamically at run time and on a per-enclave basis.

5 Design of the CURE Architecture

CURE provides a novel design that addresses the requirements described above and provides a TEE architecture with strongly-isolated and highly customizable enclaves, which can be adapted to the requirements of the services they protect. Unlike other TEE architectures, which only provide a single enclave-type, CURE allows to freely define enclave boundaries and thus, different enclaves can be constructed, as shown in Figure 2. First, in Section 5.1, we describe the ecosystem around CURE. Then, we elaborate on the different enclave types in Section 5.2. CURE's key component enabling this flexible enclave construction is its enclave ID-based access control in the system bus which manages all per-enclave resource mappings, e.g, peripherals or main memory, indicated by the different background patterns in Figure 2 and Figure 3. Our hardware primitives are presented in Section 5.3.

5.1 CURE Ecosystem

The ecosystem around CURE consists of device vendors which produce the devices implementing CURE, device users and service providers. Some services contain sensitive data (from the users and/or the service provider) and thus, must be protected. In CURE, sensitive services are either split into a sensitive and a non-sensitive part, which get included into an enclave and an user-space app (called host app), respectively, or alternatively, integrated entirely into an enclave, requiring only minimal modifications at the service. In the later case, the host app is only needed to trigger the enclave. Initially, the enclave binary does not contain sensitive data.

For every enclave, the service provider creates a configuration file which contains the enclave's requirements regarding system resources (e.g., memory, caches or peripherals), a version number and an enclave label L_{encl}. Enclave binary, configuration file and host app are bundled and deployed by the service provider over an app store (e.g., Google Play Store)

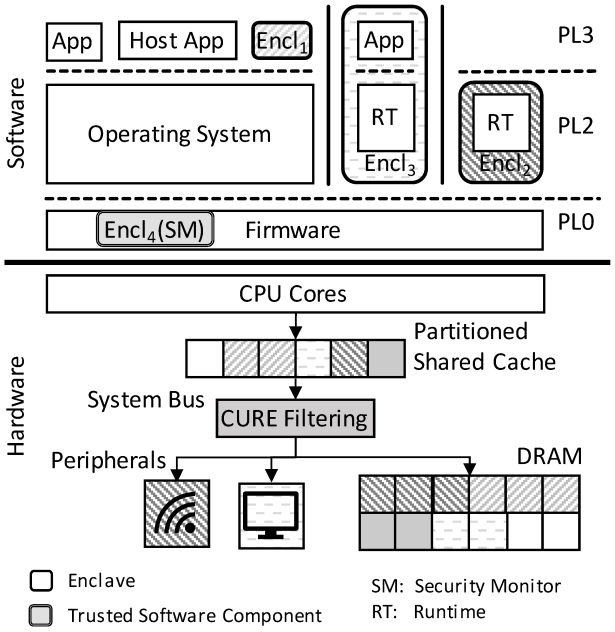


Figure 2: CURE privilege levels and enclave types, namely, user-space enclaves (Encl_1), kernel-space enclaves (Encl_2, Encl_3) and sub-space enclaves (Encl_4).

which is operated by a third party (e.g., Google). The label L_{encl} is globally unique in the app store.

Every service provider creates an asymmetric key pair SK_p and PK_p, and a public key certificate $Cert_p$, which is signed by the app store operator. Using the secret key SK_p, the service provider signs the enclave binary and configuration file (Sig_{encl}) and attaches it, together with $Cert_p$, to the app bundle. $Cert_p$ can later be used on the device to verify Sig_{encl}. For this, a certificate chain $Chain_p$ up to the root certificate of the app store operator must be present on the device. When the service provider wants to update an enclave, a new signature must be created and the version number in the configuration file updated which prevents rollbacks to older (possibly flawed) versions of an enclave [103].

A device vendor creates a unique asymmetric key pair SK_d and PK_d for each device, which is provisioned to the device during production, and a public key certificate $Cert_d$ signed by the device vendor which can later be used to prove the legitimacy of the device in a remote attestation scheme. For this, the service provider must obtain a certificate chain $Chain_d$ up to the root certificate of the device vendor. When a device was compromised, $Cert_d$ can also be revoked.

5.2 Customizable and Resilient Enclaves

CURE supports enclaves that protect user-space processes (Encl_1), run in the kernel space (Encl_2) or span the kernel and user space (Encl_3). However, an enclave does not necessarily include all code of a privilege level, e.g., an enclave can only comprise parts of the firmware code (Encl_4).

5.2.1 Enclave Management

Before describing the different enclave types supported by CURE, we give an overview on CURE's enclave management.

Security monitor. All CURE enclaves are managed by the software TCB, called *Security Monitor (SM)*, as in other TEE architectures [22, 48]. As indicated in Figure 2, the SM itself represents an enclave which is part of the firmware. As described in Section 2, we assume a system that performs a secure boot on reset, verifies the firmware (including the SM) and then jumps to the entry point of the SM. Further, we assume that the SM has already loaded its rollback protected state S_{sm} into the volatile main memory. The SM state contains SK_d, PK_d, $Cert_d$, $Chain_p$ and a structure D_{encl} for each enclave installed on the device.

Enclave installation. When an enclave is deployed to the device, the SM first verifies the signature Sig_{encl} using $Cert_p$ and $Chain_p$. Then, the SM creates a new enclave meta-data structure D_{encl} and stores L_{encl}, Sig_{encl} and $Cert_p$ in it. Moreover, the SM creates an enclave state structure S_{encl} which is used to persistently store all sensitive enclave data. The SM also creates an authenticated encryption key K_{encl} which is used to protect the enclave state when it is stored to disk or flash memory. K_{encl} and S_{encl} are also stored in D_{encl}. Initially, S_{encl} only contains an authenticated encryption key K_{com} created by the SM, which is used by the enclave to encrypt and integrity protect data communicated to the untrusted OS, and a monotonic counter. The enclave meta-data structure D_{encl} also contains a monotonic counter used to rollback protect the enclave state.

Enclave setup & teardown. The setup of an enclave is always triggered by the corresponding host app. After the OS loads the enclave binary and configuration file, it performs a context switch to the SM. The SM identifies the enclave by the label L_{encl} and begins the enclave setup by (1) configuring the hardware security primitives (Section 5.3) such that one or multiple continuous physical memory regions (according to the configuration file) are exclusively assigned to the enclave in order to isolate the enclave from the rest of the system software. Since the binary and configuration file are loaded from untrusted software, their integrity must always be verified using Sig_{encl} and $Cert_p$. Assigning physical memory regions is inevitable when providing enclaves which are able to execute privileged software (kernel-space enclave), since this allows the enclave to control the MMU. Thus, virtual memory cannot be used to effectively isolate the enclave. (2) After enclave verification, the SM configures the hardware primitives to assign also the rest of the system resources, e.g., cache or peripherals, to the enclave according to the configuration file. All assigned resources are also noted in D_{encl}. Moreover, the SM assigns an identifier to the enclave which is stored in D_{encl} and which is unique for every enclave currently active on the device. The SM can manage up to N (implementation defined) enclaves in parallel. We provide more details on the meaning of the enclave identifier in Section 5.3. (3) In the last step, the enclave state S_{encl} is restored, i.e., loaded from disk or flash memory, decrypted and verified using K_{encl}, and then copied to the enclave memory such that it is accessible during enclave runtime. The SM also checks that the monotonic counter in S_{encl} matches the counter stored in D_{encl}.

The SM configures all interrupts to be routed to the SM while an enclave is running. Thus, the SM fully controls the context switches into and out of an enclave. While the SM is executed, all interrupts on the CPU core executing the SM are disabled. All other cores remain interrupt responsive. In CURE, hardware-assisted hyperthreading is disabled during enclave execution to prevent data leakage through resources shared between the hardware threads. Alternatively, all hardware threads of a CPU core could also be assigned to the enclave if the enclave code benefits from parallelization. In the reminder of the paper, we assume that hyperthreading is disabled during enclave runtime.

After the setup is complete, the SM jumps to the entry point of the enclave. During the enclave teardown, which can be triggered by the host app or the enclave itself, the SM securely stores the enclave state (using K_{encl}), while incrementing the monotonic counters in S_{encl} and D_{encl}, removes all enclave data from the memory and caches and reconfigures the hardware primitives.

Enclave execution. At run time, enclaves can access services provided by the SM over its API, e.g., to dynamically increase the enclave's memory or to receive an integrity report which the SM creates by signing Sig_{encl} with SK_d and by attaching $Cert_d$. The integrity report is then send to the service provider by the enclave. Subsequently, using $Chain_d$, the service provider can perform a remote attestation of the enclave. Only if the attestation succeeds, the service provider provisions sensitive data to the enclave. More complex remote attestation schemes [61] could also be implemented.

Enclaves might use services of the untrusted OS which do not require access to the plain sensitive enclave data, e.g., file or network I/O. For those cases, an enclave can utilize K_{com}, which is part of S_{encl}, to protect its sensitive data. CURE also allows multiple enclaves to share encrypted sensitive data over the OS. However, the required key exchange is assumed to be performed over the back ends of the service providers and thus, out-of-scope for CURE.

Every enclave which includes a cryptographic library can also create own keys (apart from K_{com}) and store them in S_{encl}. Thus, enclaves can also implement key rotation, revocation or recovery schemes which is, however, the responsibility of the service provider and thus, out-of-scope for CURE.

On every enclave setup/teardown and context switch in and out of an enclave, the SM flushes all core-exclusive cache resources, i.e., the data cache, the TLB and the BTB, thereby preventing information leakage across execution contexts.

5.2.2 User-space Enclaves

User-space enclaves ($Encl_1$ in Figure 2) comprise a complete user-space process.

OS integration. The key characteristic of a user-space enclave is its tight integration into the OS, i.e., it relies on the OS for memory management, exception/interrupt handling and other services provided through syscalls (e.g., file system or network I/O). The OS schedules user-space enclaves like normal user-spaces processes, only that the context switches in and out of the enclave are intercepted by the SM. The OS's services are used by all user-space enclaves which prevents code duplication. Moreover, user-space enclaves do not contain management software, leading to smaller binaries.

Controlled side-channel defenses. In controlled side-channel attacks, the adversary gains information about an enclave's execution state by observing usage of resources managed by the OS, predominantly page tables [65, 92, 101]. CURE defends against these attacks by moving the page tables of user-space enclaves into the enclave memory. More subtle controlled side-channel attacks exploit the fact that the enclave's interrupt handling is performed by the OS [91]. CURE also mitigates these attacks by allowing each enclave to register trap handlers to observe its own interrupt behavior, and act accordingly if a suspicious behavior is detected [15, 79].

Limitations & usage scenarios. A user-space enclave cannot run higher-privileged code, e.g., device drivers. Thus, all sensitive data shared with a peripheral has to be processed by drivers in the untrusted OS and thus, is unprotected if not encrypted. Hence, user-space enclaves are unable to protect sensitive services which interact with devices like sensors or GPUs. Instead, user-space enclave are beneficial when protecting short-living services that can rely on encrypted data transmission, e.g., One Time Password (OTP) generators, payment services, digital key services and many more.

5.2.3 Kernel-space Enclaves

Kernel-space enclaves can comprise only the kernel space ($Encl_2$), or the kernel and user space ($Encl_3$).

Providing OS services. The key characteristic of a kernel-space enclave is its capability to run code bare-metal on a CPU core in the privileged (PL2) software layer or even in the hypervisor level (PL1) if available. Thus, OS services, e.g. memory management, can be implemented inside the enclave in a runtime (RT) component (Figure 2). This results in less resource sharing with the untrusted OS, and thus, it is easier to protect against controlled side-channel attacks [91, 92, 101]. Moreover, by including device drivers into the RT, a secure communication channel to peripherals can be established. Furthermore, kernel-space enclaves provide more computational power since CURE allows to run kernel-space enclaves across multiple cores. In CURE, peripherals can either be assigned exclusively to a single enclave, by the SM, at enclave setup or shared between different enclaves and/or the OS. The peripheral's internal memory is flushed by the SM when (re-)assigned to a new entity to prevent information leakage [49, 72, 107].

Protecting virtual machines. CURE's ability to include the kernel space into the enclave allows the construction of enclaves that encapsulate complete virtual machines (VMs). VMs are not self-contained but rely on memory and peripheral management services provided by a hypervisor, which makes the VM enclave vulnerable to controlled side-channel attacks [38, 51]. CURE mitigates this by moving the VM page tables into the enclave memory and including unmodified complete drivers into the enclave to avoid dependencies on the untrusted hypervisor [16, 17]. As for other kernel-space enclaves, peripherals are temporarily assigned to VM enclaves by the SM. Again, before a peripheral is reassigned, its internal memory is sanitized by the SM.

Limitations & usage scenarios. Sensitive services can be ported to kernel-space enclaves without changing them. However, in contrast to user-space enclaves, an enclave RT needs to be added which increases the binary size, adds development overhead and increases the memory consumption. Moreover, the CPU cores selected for the enclave first have to be freed from pending processes, detached from the OS and the RT booted on them. Nevertheless, kernel-space enclaves are required when protecting services which heavily rely on peripheral communication, e.g., authentication services using biometric sensors, ML services collecting input data over sensors or offloading computations to accelerators, DRM services or in general services which require secure I/O.

5.2.4 Sub-space Enclaves

In CURE, enclave trust boundaries can be freely defined which allows to construct fine-grained enclaves that only include parts of the software residing in a privilege level, therefore called sub-space enclaves.

Shrinking the TCB. Sub-space enclaves are especially appealing when constructed in the highest privilege level (PL0) of the system ($Encl_4$ in Figure 2). In CURE, sub-space enclaves are used to isolate the SM from the firmware code to protect against exploitable memory corruption vulnerabilities that might be present in the firmware code [24]. Moreover, hardware countermeasures, described in Section 5.3, are used to prevent the firmware code from accessing the SM data or hardware primitives. Ultimately, this minimizes the software TCB in CURE, as opposed to other TEE architectures that rely on a software TCB containing all code in the highest privilege level, i.e., EL3 (ARM) or the machine level (RISC-V), e.g., TrustZone [3], Sanctuary [10], Sanctum [22], Keystone [48].

5.3 Hardware Security Primitives

To provide CURE's customizable enclaves, new security primitives (SP) are needed in hardware. Our SPs augment the

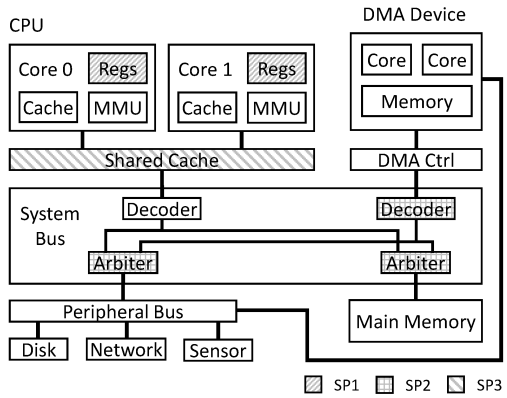


Figure 3: CURE Security Primitives (SPs), added at core register files (SP1), system bus (SP2) and shared cache (SP3).

register file of each CPU core (SP1), the system bus (SP2) and the shared cache (SP3). Figure 3 shows where CURE's SPs integrate in a modern system as assumed in Section 2.

5.3.1 Defining Enclave Execution Contexts (SP1)

Enclave ID register. In CURE, enclave execution contexts are defined using IDs, which are saved in a register that is added to every CPU core of the system (SP1). At any point in time, the value of this register, called `eid` (enclave ID) register, indicates which enclave a core currently executes. The `eid` registers are set by the SM during enclave setup, teardown and any context switch in and out of an enclave, thus, enabling flexible configuration of enclave boundaries.

Whenever an enclave is set up, the SM assigns it an unused ID. In contrast to the constant enclave labels L_{encl} (Section 5.2.1) , which are globally unique, an enclave ID is only valid as long as the enclave is loaded in memory. When an enclave is torn down, the ID gets freed and can be assigned to the next enclave. Constant IDs are only assigned to the SM and all untrusted software. The number of different IDs (N) that can be stored in `eid` defines how many enclaves can run in parallel (Section 5.2.1). However, the total number of enclaves that can be deployed is not restricted.

Propagating the enclave ID. The enclave ID is propagated through the entire system and used in the SPs to perform access control on the system resources. We incorporate the enclave ID in the bus protocol between the CPU, shared cache and system bus. In protocols like AMBA AXI4/ACE [54], the de facto on-chip communication standard, no protocol extensions are required since the bus channels provide optional user-defined signals which can be utilized to transmit the enclave ID in bus transactions. In our CURE prototype, we extend the TileLink protocol [80] by an enclave ID signal, which we describe in more detail in Section 6.

5.3.2 Access Control on the Bus (SP2)

In order to isolate enclaves and assign peripherals to them, access control mechanisms need to be implemented in hardware. As described in Section 2, the system bus represents the central gateway of a computer system that connects bus parents (CPU or DMA devices) with bus childs (peripherals or the main memory) and routes all their transactions. CURE leverages this centralization and further extends it to perform access control on parent-child transactions (SP2 in Figure 3). Incorporating carefully crafted access control at the system bus, with latency and performance in mind, reduces the overall hardware costs significantly.

Enclave memory isolation. One key task of a TEE architecture is enforcing strong isolation of the enclave code and data in the main memory. In CURE, this is achieved by performing access control in the arbiter logic in front of the main memory chip, as shown in Figure 3. This requires adding new registers and control logic to the already existing arbiter, which can only be configured (over MMIO) by the SM to assign memory regions to enclaves. Whenever the CPU requests access to a memory address, the arbiter uses the enclave ID signal, which is sent within the bus transaction, to verify if the enclave currently executing is allowed to access the memory region. At access violation, the memory access is prevented and an interrupt is triggered by the system bus, which is handled by the SM. Incorporating the required logic for this access control at the main memory side, instead of the CPU side, reduces the additional registers and logic required, which would otherwise be duplicated for every CPU core, as we show in Section 8.1.

Assigning peripherals to enclaves. The CPU interacts with peripherals over peripheral memory mapped to the CPU address space (MMIO). In CURE, access control on the MMIO memory is performed using registers and control logic added to the arbiter at the peripheral bus. The SM assigns the MMIO region of every peripheral either to one enclave exclusively or to multiple enclaves/OS by configuring the arbiter registers. Access control is then performed in the added hardware logic based on the enclave ID signal of a bus transaction. Incorporating this logic at the CPU side would have increased the hardware costs because of per-core duplication.

DMA protection. Peripherals which share large amounts of data with the CPU typically access the main memory directly over a DMA controller. CURE must protect enclaves from DMA attacks [63, 76] and also allow to assign DMA-capable peripherals to enclaves. To achieve this, CURE adds registers and control logic to the decoder in front of every DMA device. These registers define which memory regions the DMA device is allowed to access. Whenever a DMA device gets assigned to an enclave, the SM updates the device registers accordingly. Adding the required logic at the child arbiters would increase the hardware costs because enclave IDs would also need to be assigned to the DMA devices which would result in additional logic for ID comparison.

By assigning dedicated memory regions to an enclave and a DMA-capable peripheral, and by assigning the MMIO memory regions of that peripheral exclusively to the enclave, CURE

achieves an enclave-to-peripheral binding. Since neither the OS nor any other enclave can access the memory regions over which the bound enclave and peripheral communicate, no encryption or authentication schemes are required.

5.3.3 On-Demand Cache Partitioning (SP3)

CURE's enclave management (in Section 5.2.1) mitigates side-channel attacks on core-exclusive resources, such as the L1 cache, by flushing all such structures at every enclave context switch. Nevertheless, this still leaves enclaves vulnerable to cross-core attacks on the shared last-level cache [36, 39, 102]. However, vulnerability to these sophisticated attacks depends on whether the enclave code performs memory accesses dependent on sensitive data. While algorithms and implementations can be constructed leakage-resilient [2, 68], this is not directly applicable to any given application code, and thus, we provide on-demand per-enclave cache partitioning in CURE.

Security guarantees for cache side-channel resilience can be provided in hardware by either enforcing strict partitioning of resources across the different enclaves [42, 58, 97] or deploying randomization-based cache schemes [59, 60]. Nevertheless, these schemes either reduce the cache resources available for an enclave or incur additional access latency. This results in an inevitable performance overhead on the protected as well as unprotected software. The additional security guarantee, along with its resulting performance cost, is not usually required for all enclaves and largely depends on the use case.

Thus, CURE addresses these diverse enclave requirements and incorporates on-demand way-based partitioning of the shared cache (SP3 in Figure 3). This allows that cache partitioning is enabled and configured individually and dynamically for each enclave at setup and runtime. Each cache way can be allocated exclusively to an enclave. Access control on the enclave ID signal of the memory access transaction is used to permit the enclave to access (read/write or even evict) a cache way, thus ensuring strict isolation. However, when this cache isolation is not enabled for an enclave, only read/write access control on the owner enclave of each cache line is performed. This defends against a privileged adversary that can access cached enclave memory by mapping it into its own address space. As each cache line is owned by a single enclave at any point in time, access control on cache lines corresponding to shared memory between enclaves and the OS is a challenge. To address this, the SM flushes relevant cache lines at context switches between an enclave and the OS while managing shared-memory communication.

We deploy way-based partitioning because it is the least extensive in terms of hardware modifications. However, CURE provides the necessary infrastructure and mechanisms (by identifying each enclave and propagating this throughout the system bus and shared cache) to incorporate more sophisticated side-channel-resilient cache designs [25, 74, 99].

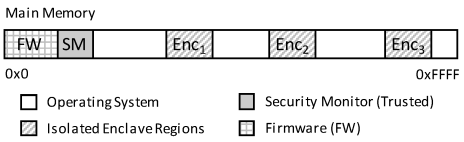


Figure 4: Physical memory layout of our CURE prototype.

6 Prototyping CURE on RISC-V

While CURE is architecture-agnostic and can be ported to other ISAs, we prototype it here for a RISC-V system based on the open-source Rocket Chip generator [4]. We describe next our CURE instantiation, followed by details on the implemented enclave types and hardware security primitives.

RISC-V System-on-Chip platform. We build a RISC-V System-on-Chip (SoC) using the Rocket Chip generator [4]. For prototyping, we equipped the SoC with multiple in-order Rocket cores, in line with prototyping efforts in related work [22]. Each Rocket core has one hart (representing a hardware thread), an own MMU, BTB, TLB and L1 cache. The SoC also contains a system bus which connects the cores to system peripherals (over the peripheral bus) and system main memory. We integrate a shared L2 cache [81] between the system bus and the main memory. A DMA device is connected to the system bus as a bus parent. As a result, this SoC resembles our assumed platform shown in Figure 3, except that the L2 cache is integrated as a last-level cache after the system bus.

We implement our prototype on this SoC aiming to maintain minimum hardware and no additional latency. We use 4 bits to represent the enclave ID, i.e., our prototype can distinguish 16 (N) enclaves, where ID `0` is statically assigned to the OS, ID `0xF` to the Security Monitor (SM) and ID `0xE` to the firmware (explained in Section 6.2.2). The remaining 13 IDs can be freely assigned to enclaves. We assign one continuous physical memory region to each enclave, resulting in the memory layout shown in Figure 4. We choose to assign only one region per enclave to simplify our prototype and minimize the induced hardware overhead. The CURE design, however, also allows for multiple continuous regions per enclave. The SM and firmware memory regions are adjacent since they are both deployed as part of the bootloader [29]. All regions not assigned to an enclave, SM or the firmware, belong to the OS. Supporting more enclaves in parallel is possible if the additional hardware overhead is acceptable.

Software stack. The Rocket core supports three software privilege levels (user, supervisor and machine). Hypervisor support is still a work-in-progress [28] and thus, we do not consider it in our prototype. In the supervisor level, we use an OS consisting of a modified Linux LTS kernel 4.19 with a Busybox 1.29.3 environment. We add a custom kernel module which performs security-uncritical tasks during the enclave setup. We implement the SM in the machine level as a sub-space enclave to separate it from the firmware which runs in the same privilege level.

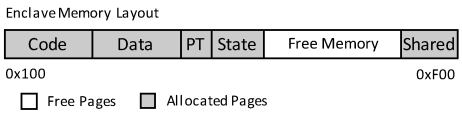


Figure 5: CURE enclave memory layout consisting of the code & data pages, page tables (PT), the enclave state (State) and the shared memory (Shared).

Cryptographic underpinnings. In the implemented CURE prototype, we use Ed25519 [71] as the digital signature scheme for the signing and verification of the enclave signature Sig_{encl} and the integrity report used for remote attestation, as described in Section 5.2.1. Thus, SK_d/PK_d and SK_p/PK_p are Ed25519 key pairs. The public key certificates $Cert_d$ and $Cert_p$ are implemented in the X.509 format. In our CURE prototype, the certificate chains $Chain_d$ and $Chain_p$ required to verify $Cert_d$ and $Cert_p$ are, for the sake of simplicity, represented by two Ed25519 public keys. As described in Section 5.2.1, $Chain_p$ is included in the SM, whereas $Chain_d$ is required at the service provider. The enclave state S_{encl} and enclave data communicated with the OS are protected through authenticated encryption, using the keys K_{encl} and K_{com}, respectively. We use AES-GCM from libtomcrypt 1.18.2. [52] as the authenticated encryption scheme and include it in the SM. Moreover, we also add it to our implemented enclaves, such that the enclaves can create additional keys. Consistent with Section 5.2.1, the SM holds a metadata structure D_{encl} for each enclave which contains $Cert_p$, Sig_{encl}, K_{encl} and S_{encl}, whereas K_{com} is part of S_{encl}.

6.1 Software CURE Enclaves

Our CURE prototype implements user-space enclaves, kernel-space enclaves and sub-space enclaves and thus, fulfills requirement `FR.1` (Section 4.2). In the following, we describe the enclave memory layout and give implementation details on each enclave type.

6.1.1 Enclave Memory Layout

In our prototype, each enclave is assigned a continuous physical memory region which is allocated during enclave setup using Linux's Contiguous Memory Allocator (CMA). The enclave memory layout is shown in Figure 5. At the lowest address, the enclave code and data pages are loaded by the OS. The enclave page tables are only stored in the enclave memory while the memory management is performed by the untrusted OS. During the enclave setup, the SM loads the enclave state S_{encl} into the enclave memory. The free memory space is used for dynamic memory allocation. The memory region at the highest address is used for the communication between enclave and OS. Since our prototype allows one continuous memory region per enclave, the shared memory region is either assigned to the communicating enclave or to no enclave, which automatically assigns the region to the OS. When the enclave is set up, the address of the shared memory region is communicated to the OS via the return value of the SM call. The enclave is informed by storing the address information on the stack of the enclave. The size of the enclave state and shared region can be freely set, we set them to 64 bytes and 4 KB, respectively.

6.1.2 Security Monitor

We implement the SM as a sub-space enclave (Enc_5 in Figure 2) separated from the firmware in memory (Figure 4), which is enforced by the hardware security primitives. However, this leaves the firmware with access to the security-critical machine level registers `eid`, which we added, and `mtvec`, which holds the base address of the trap vector that the core jumps to after an interrupt. To prevent the firmware from configuring these registers, we implement a hardware mechanism that ensures that the `eid` and `mtvec` registers can only be written to when the `eid` register is set to the SM ID (`0xF`). The `eid` register is, in turn, set to `0xF` by the hardware when performing a context switch to machine mode that traps in the SM.

6.1.3 User-space Enclaves

Memory management. Since the memory management of the user-space enclave (Enc_1 in Figure 2) is performed by the untrusted OS, we include the enclave page tables in the enclave memory, to prevent page table based attacks [65, 92, 101]. During enclave setup, the OS creates the page tables exactly as for a normal process. However, the OS turns off demand paging and maps all code and data pages to prevent page faults during enclave execution. The page tables are then handed to the SM which verifies their validity. Moreover, the SM verifies that the supverisor address translation and protection (`satp`) register, which holds the address of the root page table, points into the enclave memory. Subsequently, the page tables are copied to the enclave memory. Once the enclave is setup, the OS cannot alter the page tables anymore. When the dynamic allocation of memory leads to a page fault, the OS creates a new page table entry and passes it to the SM which includes it into the page tables.

Syscalls. Our prototype provides enclaves which can use OS services, e.g., file or network I/O, over Linux syscalls which trap in the SM. We include AES-GCM into the enclaves to encrypt and integrity-protect sensitive data shared with the OS, using K_{com}. Enclaves are always exited through the SM which is enforced by clearing the machine exception delegation (`medeleg`), machine interrupt delegation (`mideleg`), supervisor exception delegation (`sedeleg`) and supervisor interrupt delegation (`sideleg`) registers during enclave setup. During run time, the enclave can register custom trap handlers which are called by the SM before switching to the

OS after an interrupt. Thus, the enclave can observe its own interrupt behavior and detect suspicious behavior caused by interrupt-based side-channel attacks [15, 91].

6.1.4 Kernel-space Enclaves

Our CURE prototype supports kernel-space enclaves with and without user space (Enc_3 and Enc_2 in Figure 2). We use an Linux LTS kernel 4.19, which currently on RISC-V does not support a suspension mode, as the enclave RT.

Allocating resources. When an enclave is set up, the custom kernel module unmounts the driver modules of all peripherals requested by the enclave. Then, the SM performs the security-critical tasks of the enclave setup, as described in Section 5.2.3. When the enclave binary is successfully verified, the kernel module shuts down the core(s) reserved for the enclave using the Linux hotplugging mechanism. Next, a switch to the SM is performed which jumps to the entry point of the enclave RT in order to boot the RT on all reserved cores. At enclave shutdown, the SM performs the cleanup, and all freed cores are reintegrated into the OS. Then, the kernel module remounts the driver modules.

Enclave-OS communication. Since our CURE prototype allows one memory region per enclave, access to a shared region needs to be requested at the SM which then assigns the shared region to the requesting party (sender). Once the sender is finished accessing the shared region, the SM assigns the shared region to the receiver and notifies the receiver about new data in the shared region using an inter-processor interrupt. In contrast to the user-space enclave, only external interrupts are trapped in the SM during kernel-space enclave execution which is enforced by configuring the `medeleg` and `sedeleg` registers during the enclave setup. All interrupts triggered by the enclave cores are handled by the RT.

6.2 Hardware Security Primitives

We describe next, how we modify the Rocket Chip to implement CURE's hardware security primitives (Section 5.3).

6.2.1 Extending the TileLink Protocol

We modify the Rocket core such that on every memory access, the `eid` register value is sent as part of the issued bus transaction. This also includes transactions issued by the PTW (page table walker) during the page table walk when performing address translations. Thus, if a malicious enclave modified its own page tables to point to a memory region outside of the enclave memory, the PTW transactions are blocked by the access control mechanisms on the system bus.

TileLink [80] is the default bus protocol used on the Rocket Chip to connect on-chip components. TileLink specifies five channels (`A` - `E`). When connecting a parent to the system bus which contains an internal cache, all five channels are needed

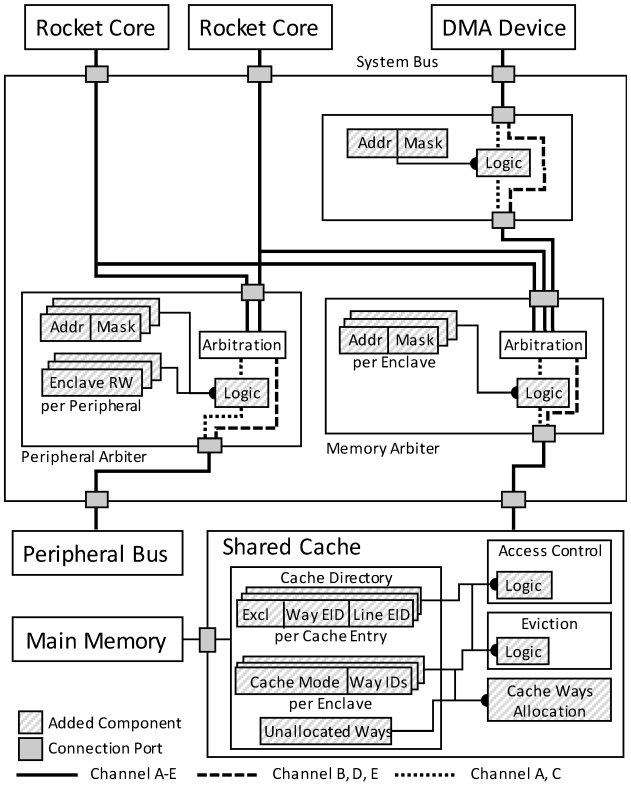


Figure 6: CURE prototype implementation using Rocket Chip.

to implement the TileLink coherence protocol (TL-C). When a parent does not require cache coherency, only the `A` and `D` channels are needed (TL-UL/UH). In our RISC-V SoC, the Rocket cores and the DMA devices are connected over TL-C since they contain L1 caches.

We extend the TileLink protocol by a 4-bit `eid` signal to propagate the enclave ID. The `eid` signal is only added to the `A` and `C` channels which transport the memory read and write transactions from the parents (CPU and DMA devices) to the system bus and childs (peripherals and main memory), respectively. All other channels remain unmodified.

6.2.2 System Bus Access Control

We implement CURE's access control mechanisms in the system bus by adding registers and control logic at the memory and peripheral arbiters and the ports connecting DMA devices. The hardware changes are shown in Figure 6, exemplary for a system containing two cores, one DMA device and multiple peripherals. All newly added components are connected to the control bus of the system and thus, are configurable by the SM over MMIO. We omit the control bus in Figure 6 for the sake of clarity. Our implementation supports enclave-to-peripheral binding and thus, fulfills `FR.4`. Moreover, in contrast to related work [20, 23], all access control is performed in parallel to arbitration, thus, guaranteeing execution in a single clock cycle without incurring additional latency.

Performing access control. The added registers hold memory ranges defined by a 32-bit base address (`Addr`) and a 32-bit mask (`Mask`), and are used by the control logic to perform access control on every memory transaction using the `eid` and `address` signals. Access control is only performed on channels with a parent-to-child direction (channels `A` and `C`). At access violation, the transaction is redirected (with all-zero data) to an unused, zero-initialized memory region. Thus, all forbidden transactions write/read zeros to/from the unused memory region. An adversary enclave might fill L1 with malicious data which could get flushed with SM privileges during enclave context switch. To prevent this, we modify the core such that on every switch to the SM, the L1 is flushed before the `eid` register is set. We connect the system bus to the peripheral and interrupt bus. This allows the SM to configure the added registers and control logic, and trigger an interrupt upon access violation which is handled by the SM.

Memory arbiter. We add 15 registers to the memory arbiter, one for each enclave (13), the SM and the firmware. Each register defines the memory region assigned to each execution context. For the enclaves, the control logic verifies that transactions only target the assigned region. For the SM, no access control is performed. The OS is allowed to access all regions except the ones specified in registers of the arbiter. The firmware is allowed to access its own and the OS regions which is why a static ID needs to be assigned to the firmware.

Peripheral arbiter. We add two registers per peripheral to the arbiter of the peripheral bus. One covers the MMIO region of the peripheral, and the other 32-bit register contains a bitmap that defines read and write permissions for every enclave.

DMA port. We add a register at every port which connects a DMA device. In CURE, a DMA device is exclusively assigned to a single enclave at one point in time. In our prototype, a DMA device accesses the main memory but not other peripherals. If specific use cases, e.g. PCI peer-to-peer transactions [67], must be supported, additional registers need to be added to specify multiple allowed memory regions. Together with the peripheral arbiter, this fulfills FR.2.

6.2.3 L2 Cache Partitioning

For cache side-channel resilience, we implement way-based flexible cache partitioning for the shared L2 (last-level) cache [81] in our prototype. We leverage the `eid`-extended TileLink memory transactions to detect when an enclave issues a cache request.

Configurable partitioning. We implement two modes of partitioning to allow enclaves to individually enable cache side-channel resilience. The first mode `CP-BASIC` performs rudimentary access control where each enclave is only permitted to access (hit) its own cache lines, but is free to evict cache lines from other ways. The second mode `CP-STRICT` provides more stringent security guarantees by allocating *exclusively* one or more ways (across all cache sets) to the pertinent enclave. Only these cache ways can be accessed by the enclave to store or evict cache lines. This provides strict isolation between the cache resources of the different enclaves, thus, effectively blocking cache side-channel leakage, but reduces the cache resources available for the enclave. Depending on the enclave service requirements, the partitioning mode can be configured by the SM independently for each enclave at setup and during the enclave lifetime, thus, fulfilling `FR.5`.

Access control. We extend each cache entry metadata with a 4-bit `line-eid` register encoding the owner enclave of the cache line, as shown in in Figure 6. We extend the cache lookup logic to generate a hit only when both tag as well as `eid` match for `CP-BASIC`, as opposed to usual tag matching.

To support `CP-STRICT`, the cache ways directory is also extended with a 1-bit register `excl` that identifies whether each way is owned exclusively by an enclave, as well as a 4-bit `eid` register that identifies the owner enclave. The cache controller logic is augmented with a register-based lookup table that is indexed by the `eid`. It encodes with a single `mode` bit whether the corresponding enclave has `CP-STRICT` enabled and its allocated cache way indices. In `CP-STRICT`, cache hits are only allowed in these cache ways.

Eviction and replacement. The L2 cache we use implements a pseudo-random replacement policy where any way is selected pseudo-randomly for eviction. We modify this to only select a way from the subset of ways allowed for each enclave. For enclaves with `CP-STRICT`, only ways exclusively allocated to it are used. For enclaves with `CP-BASIC`, all ways (except ways allocated exclusively to other enclaves) are used.

Per-enclave cache allocation. Unallocated way indices are maintained in a register vector. If an enclave with `CP-STRICT` enabled requests to exclusively own cache ways, the required ways are allocated if available and below the allowed maximum per enclave.

An inherent drawback of this partitioning technique is how the limited number of cache ways directly constrains the number of simultaneous enclaves that can have `CP-STRICT` enabled. However, this is only an implementation decision for our particular prototype, where more sophisticated cache designs [25, 74, 99] can be integrated into CURE.

7 Security Considerations

To protect from a strong software adversary, our instantiation of CURE must fulfill the security requirements introduced in Section 4.1. In the following section, we discuss how our prototype meets the requirements `SR.1`, `SR.2`, and `SR.4`, whereas we show the fulfillment of `SR.3` in Section 8.

7.1 Hardware Security Primitives (`SR.2`)

The enclave protection is enforced by hardware SPs at the system bus and L2 cache which are configured over MMIO.

After the system is powered on and on every switch to the machine level, the CPU jumps to the trap vector whose address is stored in the `mtvec` register. The trap vector is included into the SM such that the boot process and context switches are overlooked by the SM. The `mtvec` register is assigned to the SM by coupling the access permission to the SM enclave ID (stored in the `eid` register) which is also assigned to the SM. The `eid` register is set by hardware during the context switch into the machine level. During boot, the SM assigns the SP MMIO regions exclusively to its own enclave ID.

7.2 Enclave Protection (SR.1)

At rest, the enclave binaries are stored unencrypted in memory. However, during enclave setup, the SM verifies the binaries using digital signatures. Moreover, the L1 is flushed during setup/teardown to remove malicious or sensitive data from the cache. The communication between enclaves and the OS is controlled by the SM, so is the delegation of the shared memory address. Hardware-assisted hyperthreading is disabled during enclave execution. The enclave state, which is loaded during the setup process, is persistently stored by the SM using authenticated encryption, either in RAM as part of the SM state or evicted to flash/disk, and additionally rollback protected. During teardown, the SM removes all enclave data from the memory.

The SPs in hardware perform access control on physical addresses at the system bus. Thus, CURE protects from adversaries in privileged software levels (PL2 - PL0) and from off-core adversaries, e.g. peripherals performing DMA. The enclave data cached in the L1 during run time is protected by flushing it on all context switches. Data in the L2 cache is protected by assigning cache lines exclusively to enclaves. Since no enclave (except the SM), has elevated rights on the system, CURE also protects from malicious enclaves.

7.3 Side-channel Attack Resilience (SR.4)

Cache side-channel attacks. Side-channel attacks which target data in core-exclusive cache resources, i.e., in the L1 [11], the BTB [50] or the TLB [31], are prevented by the SM by flushing the resources on all context switches. Side-channel attacks targeting data in the shared L2 cache [36, 39, 102] are prevented through strict way-based cache partitioning.

Controlled side-channel attacks. Side-channel attacks on user-space enclaves which target page tables [65, 92, 101] are prevented by including the page tables into the enclave memory and by mapping all enclave code and data pages before execution. The SM verifies the page tables and the base address of the root page table stored in the `satp` register. The hardware SPs prevent the page table walker (PTW) from performing forbidden memory access during the page table walk. Side-channel attacks exploiting interrupts [91] can be mitigated using trap handlers (Section 5.2.2).

CURE provides cryptographic primitives in the user-space enclaves to encrypt and integrity-protect data shared with the OS. However, using OS services over syscalls always comprises a remaining risk of leaking meta data information [2, 77] or of receiving malicious return values from the OS [13]. In user-space enclaves, these attacks must be mitigated on the application level inside the enclave, e.g., by using data-oblivious algorithms [2, 68] or by verifying the return values [13]. None of these attacks pose a threat to kernel-space enclave since all resources are handled by the enclave RT. However, on VM enclaves, the second level page tables need to be protected, as with user-space enclaves. Interrupt-based attacks can again be mitigated with custom trap handlers. No additional countermeasures are needed to protect the SM since the SM does not use a virtual address space or OS services and handles its own interrupts.

Transient execution attacks. The discovered transient execution attacks either mistrain the branch predictor [14, 43, 45], rely on information leakage [89] or malicious injections [90] on the L1 cache, or rely on resources shared when using hardware-assisted hyperthreading [12, 78, 90, 93, 94]. By disabling hyperthreading during enclave execution (or alternatively assigning all threads to the enclave) and flushing core-exclusive caches, CURE protects enclaves against the known transient execution attacks.

8 Evaluation

In the following section, we systematically evaluate our CURE prototype. First, we quantify the software and hardware modifications required to implement CURE. Next, we evaluate the performance of CURE's enclaves using microbenchmarks, and the overall performance overhead of CURE using generic RISC-V benchmark suites.

8.1 System Modifications

Component	LOC
Linux Kernel	375 (modified)
Custom Kernel Module	200
Security Monitor	544
SM Crypto-Library	2586

Table 1: Lines of code required to implement CURE. SM Crypto-Library refers to the crypto library (part of tomcrypt) included in the Security Monitor.

Software changes and TCB. Our implementation of CURE on RISC-V comprises of a slightly modified Linux LTS kernel 4.19, a custom kernel module, and our software TCB (SM). In Table 1, the lines of code (LOC) are shown for each of the components, which indicate that the software changes required to implement CURE are minimal. Moreover, the SM only consists of around 3KLOC of code, whereas most

(82.62%) of the SM code consists of cryptographic primitives. Because of its minimal size, formal verification of the SM is possible [44], thus, fulfilling SR.3. Note that since CURE isolates the SM in an own sub-space enclave, CURE can achieve a smaller TCB size than other RISC-V security architectures [22, 48, 98] which include all code in the machine level, i.e., the firmware code, in the TCB. In our implementation, the firmware code consists of 3286 LOCs. Thus, by isolating the SM in a sub-space enclave, we managed to cut the software TCB in half, where the actual management code is even less (15.56%).

Protecting a sensitive service in a user-space enclave requires to add a small custom library (10KB) to the service binary. For the kernel-space enclaves, management code (the enclave RT) must be added in addition. In our prototype, we use the Linux LTS kernel 4.19 as the RT which increases the size of the service binary by 3MB. Custom RTs can further decrease this kernel-space enclave overhead. However, kernel-space enclaves will always have an increased binary size and memory consumption compared to user-space enclaves.

Hardware overhead. We evaluate the hardware overhead of our changes by synthesizing the generated Verilog descriptions using Xilinx Vivado tools targeting a Virtex UltraScale FPGA device. Table 2 shows a breakdown of the individual area overhead of the different modifications required to implement CURE. Overhead is represented in look-up tables (LUTs), the fundamental programmable logic blocks of FPGA devices, and registers.

Configuration	LUTs Overhead (+%)	Registers Overhead (+%)
Baseline	61,097	28,012
TileLink extension	+211 (0.4%)	+110 (0.4%)
Access control extensions		
Main memory	+5,276 (8.6%)	+1,055 (3.8%)
1 MMIO peripheral	+248 (0.4%)	+107 (0.4%)
1 DMA device	+112 (0.2%)	+72 (0.3%)
On-demand cache partitioning		
w/ L2 cache (baseline)	+30,232	+11,549
w/ L2 cache partitioned	+516 (1.7%*)	+214 (1.8%*)

Table 2: Hardware overhead breakdown in LUTs and registers. Baseline setup consists of 2 Rocket cores without L2 cache. *Overhead relative to the L2 cache (baseline).

We compare in Table 2 with a baseline configuration of 2 in-order Rocket cores (each with L1 cache). Extending the TileLink protocol throughout the system bus incurs a minimal overhead of 105 LUTs per core relative to the baseline (211 LUTs for 2 cores). This overhead includes propagating the `eid` in tandem with memory access transactions through the MMU of every core, and is thus replicated for every additional core in the system.

In contrast, the rest of our modifications for performing access control at the system bus, including enclave-to-peripheral binding, are independent of the number of cores. Incorporating logic to perform access control for every MMIO peripheral utilizes an additional 248 LUTs, and 112 LUTs per DMA device. Each represent below 0.5% overhead relative to a dual-core baseline SoC. Integrating an L2 cache into our baseline setup utilizes an additional 30,232 LUTs. Applying our on-demand way-based partitioning to this cache costs only 516 LUTs and 214 registers, which is 1.8% overhead relative to the L2 cache logic utilization itself, and 0.5% relative to the entire SoC. Our area overhead evaluation results demonstrate that the hardware modifications required to achieve our fine-grained and customized enclave protection in CURE indeed incur minimal area overhead on both single- and multi-core architectures, thus fulfilling FR.3.

Measurement	Normal Process	User-Space Enclave	Kernel-Space Enclave
Setup:	0.741	23.918	413.726
Binary Verification	-	21.824	218.975
Others	0.741	2.094	194.750
Teardown:	0.065	23.531	103.517
Memory Cleaning	-	9.384	50.206
Others	0.065	14.147	53.311
Context switch to OS	0.008	0.025	53.308
Context switch from OS	0.078	0.084	194.747
Dynamic memory allocation	0.003	0.020	0.005
OS communication	-	0.020	0.049

Table 3: CURE performance overhead compared to a normal process on microbenchmarks in milliseconds.

8.2 Performance Evaluation

We evaluate the performance of CURE using our FPGA-based setup coupled with cycle-accurate simulators. We conduct our experiments using micro and macro benchmarks for user-space and kernel-space enclaves, and compare them to unmodified user-space processes. We conduct 10 runs for each of the experiments.

8.2.1 Microbenchmarks

For microbenchmarks (Table 3), we measured important key aspects individually: setting up and tearing down an enclave, context switching with the OS, dynamic memory allocation, and communication via shared memory. We implement an application which performs the required tasks (without any additional logic) and run it as a normal Linux process, a user-space enclave and a kernel-space enclave (single core). The enclave setup is triggered by a host app in Linux which is the only purpose of the app. The enclave binary sizes therefore mainly correspond to the overhead produced by the enclave types, i.e., 10KB for the user-space enclave and around 3MB for the kernel-space enclave.

For the enclave setup, our results show that most of the time (91.3% for user-space, 52.1% for kernel-space enclaves) is spent on binary verification. The *Others* measurement

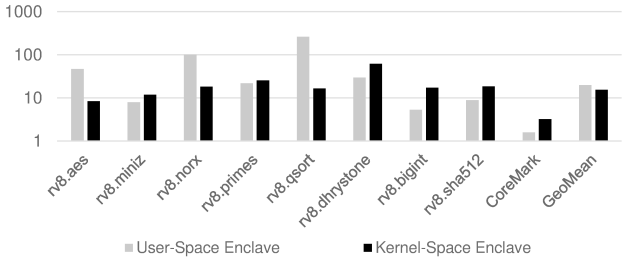


Figure 7: CURE performance overhead (in percent) on macro benchmarks `rv8` and `CoreMark` relative to a normal process.

contains all remaining steps of the setup process, e.g., loading of the enclave binary, enclave configuration, flushing of the TLB and L1 cache and jumping into the enclave. During our evaluation, we use 32KB 8-way set associative L1 data and instructions caches and a TLB with 32 entries. The setup of the kernel-space enclave is more complex and includes additional setup steps, namely, freeing the core from pending processes, detaching the core from the OS, and booting the RT. In the teardown phase, zeroing the memory produces 39.9% of the overhead for the user-space and 45,7% of the overhead for the kernel-space enclave). The cleaning is more time consuming for the kernel-space enclave because of the larger enclave memory region. The *Others* measurement contains additional steps, e.g., exiting the enclave and flushing the TLB and L1 cache. In the kernel-space enclave case, the core must additionally be rebooted.

As the RT in our prototype does not support a suspension mode (keeping the enclave in memory), we emulate the *context switch to the OS* by performing a teardown without zeroing memory, and the *context switch from the OS* by performing a setup phase without verifying the enclave binary. Suspending the enclave and restoring it should be faster than a regular shutdown and boot, thus, this represents a worst-case approximation. The context switching measurements also contain the overhead for flushing the TLB and L1 cache, for which we measure 28 cycles and 3141 cycles, respectively.

As new entries to the page tables need to be verified by the SM, user-space enclaves have a higher overhead for dynamic memory allocation. In the kernel-space enclave case, all page tables are included in the enclave memory and thus, do not need to be verified. During communication, the OS can directly access a process's memory, whereas the user-space enclave needs to copy the data to be shared to the shared memory region. The kernel-space enclave additionally has to request the shared memory from the SM, and the OS needs to be notified by the SM using an inter-process interrupt.

8.2.2 Macrobenchmarks

To evaluate the performance overhead in realistic scenarios, we used three different benchmarking suites that stress single cores, multi-core setups with two cores under test, and how the enclaves influence an OS under load. Furthermore, we measure the performance impact of our L2 cache partitioning by assigning 1/16 of the L2 cache to the enclave under test.

Single-core benchmarks. For single-core performance, we evaluated CURE with the RISC-V benchmark suites `rv8` [75] and `CoreMark` [26], which are commonly used for TEE architectures [22, 48]. The results depicted in Figure 7 are normalized to a normal user-space process. We measured a geometric mean of 19.70% for user-space enclaves and 15.33% for kernel-space enclaves for the performance overhead. As shown in Table 3, kernels-space enclaves have an increased setup time which however, amortizes with longer enclave run times. Outliers like `aes`, `norx` and `qsort` are memory-intensive workloads that perform a large number of context switches to the OS, mainly for dynamic memory allocation. Performing context switches and dynamic memory allocation is more expensive for the user-space enclave since the SM must verify newly created page table entries and copy them to the enclave memory. During one run, we count 24,601 syscalls for `aes`, 24,602 syscalls for `norx` and 48,846 syscalls for `qsort`. We also measure the overhead for flushing the TLB and L1 on every context switch which is, however, only necessary for the user-space enclave. The flushing induces only a small overhead which makes up for 1.03%, 1.48% and 1.21% of the overall overhead for `aes`, `norx` and `qsort`, respectively.

Load/Cores	Normal Process	Kernel-Space Enclave
30/1	1.49	1.49 (+-0.00%)
30/2	0.75	0.78 (+4.00%)
500/1	27.65	28.82 (+4.23%)
500/2	14.42	14.60 (+1.25%)
1000/1	56.00	55.28 (-1.29%)
1000/2	27.64	27.81 (+0.62%)
1500/1	83.62	83.64 (+0.02%)
1500/2	41.82	42.62 (+1.91%)
2000/1	111.70	111.99 (+0.26%)
2000/2	56.00	57.62 (+2.89%)
GeoMean	-	+0.9%

Table 4: Kernel-space enclave performance on multi-core `stress-ng` benchmark in seconds.

Multi-core benchmarks. Since CURE allows to assign multiple core to a kernel-space enclave, we evaluated CURE also on the dedicated multi-core benchmark `stress-ng` [41]. The results in Table 4 show that multi-core kernel-space enclaves are practical by achieving almost the same performance as normal processes.

Influence on OS. We stress the OS by running `CoreMark`, while starting an enclave in parallel. For the user-space enclave we use a single core, while two cores are needed for the kernel-space enclave, for which we simulate the suspension mode as in the microbenchmarks. For one core, the CoreMark running on the OS is slowed down by 0.519s (1.56%). For two cores with only one call after setting up the kernel-space enclave, the OS is slowed down by 0.792s (4.23%), showing

Benchmark	Cycles # for 16/16 ways (baseline)	Cycles # for 1/16 ways (worst-case)	Overhead (+%)
rv8.aes	29,754,631,670	32,175,733,155	8.1%
rv8.miniz	42,040,536,353	45,063,752,315	7.2%
rv8.norx	30,899,386,564	32,702,249,193	5.8%
rv8.primes	21,731,621,683	21,770,731,965	0.18%
rv8.qsort	24,355,792,115	25,280,228,818	3.8%
rv8.dhrystone	19,865,586,529	20,289,555,571	2.1%
rv8.bigint	65,512,466,917	71,487,944,568	9.1%
CoreMark	394,664,199	402,293,814	1.9%
GeoMean	-	-	3.09%

Table 5: Performance impact of L2 cache strict way-based partitioning for kernel-space enclaves on different benchmarks.

that the kernel-space enclave has a higher performance impact on the OS than the user-space enclave. Based on these results, we demonstrate that CURE also fulfills FR.4 and achieves a moderate performance overhead.

L2 cache partitioning. We evaluate the performance impact of partitioning the L2 cache (`CP-STRICT` mode) for kernel-space enclaves and show our results in Table 5. For our cycle-accurate experiments, we configure the core with 64KB 8-way set-associative L1 data and instructions caches and 2048KB 16-way set-associative shared L2 cache. The impact of way-based cache partitioning on performance is very application-dependent (besides the caches configuration and caches and main memory access latencies), as demonstrated by our experiments where the performance overhead ranges from a little under 0.2%, as for the `prime` benchmark, to a little over 9% for the `bigint` benchmark, for example. We measure a geometric mean of 3.09%. We note that the overheads reported are performance hits where the baseline is a best-case scenario where the only workload utilizing the cache resources (all 16 ways of the L2 cache) is the kernel-space enclave under test. Furthermore, we observe that performance significantly improves once more than 1 way is allocated per enclave, which is the likely scenario for enclaves that run applications with larger working sets and can benefit more from increased L2 cache resources.

9 Related Work

The existing works mostly related to CURE are TEE architectures which focus on modern high-performance computer systems. In contrast to capability systems or memory tagging extensions [30, 82, 88, 95, 100], TEE architectures protect sensitive services in security contexts (enclaves) against privileged software adversaries. We do not further discuss TEE architectures focusing on embedded systems [8, 47, 66, 98].

We compare CURE to other TEE architectures in Table 6. All presented architectures provide a single type of enclave which, on an abstract level, resemble either the user-space or kernel-space enclaves provided by CURE.

Intel SGX [64] offers user-space enclaves on Intel processors. The untrusted OS provides memory management and other OS services, e.g. exception handling, to the enclaves. SGX does not protect against cache side-channel [11, 50] and controlled side-channel attacks [91, 92, 101]. Many extensions to SGX were proposed in order to mitigate side-channel attacks [1, 2, 7, 15, 69, 79], however, these solutions are all ad-hoc approaches that do not fix the underlying design shortcomings of SGX, but instead leverage costly data-oblivious algorithms [1, 2, 7], or exploit not commonly available hardware in an unintended way [15, 79].

Sanctum [22], which also provides user-space enclaves, addresses both, cache side-channels through page coloring, and controlled side-channels by storing the enclave page tables in the enclave memory, like CURE. However, page coloring is not practical as it influences the whole OS memory layout and cannot be efficiently changed at run time. CURE's cache partitioning instead allows dynamic assignment of cache ways, and also mechanisms to mitigate interrupt-based side-channel attacks. Sanctum and SGX only provide user-space enclaves which are inherently limited as they cannot provide secure I/O, but only protect from simple DMA attacks.

Similar to SGX, AMD SEV [38], which isolates complete VMs in the form of kernel-space enclaves, does not consider any side-channel attacks. VM data in the CPU cache is protected by an access control mechanism relying on Address Space Identifiers which, however, does not protect against cache side-channel attacks. As the memory management and I/O services are provided by the untrusted hypervisor, SEV is also vulnerable to controlled side-channel attacks [65] and cannot provide secure peripheral binding [51].

ARM TrustZone [3] separates the system into normal and secure world, a single kernel-space enclave which does not rely on the OS and thus, is protected from controlled side-channel attacks. TrustZone does not provide cache side-channels protection, only by using additional hardware [106]. Further, TrustZone's major design shortcoming is providing only a single enclave, thus, sensitive services cannot be strongly isolated with TrustZone, hence, access to TrustZone is highly limited in practice by device vendors. Extensions building upon TrustZone mostly tried to enable multi-enclave support for TrustZone [10, 18, 33, 85] with workarounds that either rely on ARM IP [10], block the hypervisor [18, 33], or massively impact performance [85]. Since multiple enclaves were not considered in the TrustZone design from the beginning, even the proposed extensions cannot provide binding peripherals directly and exclusively to single enclaves.

Keystone [48] provides kernel-space enclaves on RISC-V. Moreover, Keystone uses a cache-way based partitioning against cache side-channel attacks, comparable to CURE. However, Keystone provides a coarse-grained cache ways assignment per CPU core, whereas CURE assigns cache ways to enclaves with freely configurable boundaries. Thus, the Keystone design is limited to a single enclave type which prevents Keystone from isolating the firmware from the actual TCB and demands adapting the sensitive services to the

Name	Extensions	Enclave Type			Dynamic Cache Side-Channel Resilience	Controlled Side-Channel Resilience	Enclave-to-Peripheral Binding
		User-Space	Kernel-Space	Sub-Space			
SGX [64]	[1, 2, 7, 15, 69, 79]	●	○	○	◐*	◐*	○
Sanctum [22]	-	●	○	○	◐	●	○
SEV(-ES) [38]	-	○	●	○	○	○	○
TrustZone [3]	[10, 18, 27, 32, 33, 57, 85, 106]	○	●	○	◐*	●	◐
Keystone [48]	-	○	●	○	●	●	○
CURE	-	●	●	●	●	●	●

Table 6: Comparison of major TEE architectures with respect to provided enclave types, dyn. cache-side channel and controlled-side channel resilience, and enclave-to-peripheral binding, i.e., MMIO/DMA protection with exclusive enclave assignment. ● indicates full support, ◐ for support with limitations, ○ for no support, * if resilience can only be achieved through extensions.

predefined enclave. Moreover, in contrast to CURE, Keystone does not support enclave-to-peripheral binding.

10 Conclusion

We presented CURE, a novel TEE architecture which provides strongly-isolated enclaves that can be adapted to the functionality and security requirements of the sensitive services which they protect. CURE offers different types of enclaves, ranging from sub-space enclaves, over user-space enclaves, to self-sustained kernel-space enclaves which can execute privileged software. CURE's protection mechanisms are based on new hardware security primitives on the system bus, the shared cache and the CPU. We instantiate CURE on a RISC-V system. The evaluation of our prototype indicates minimal hardware overhead for the security primitives and a moderate overall performance overhead.

Acknowledgments

We thank our anonymous reviewers for their valuable and constructive feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297. Moreover, this project has received funding from Huawei within the OpenS3 lab.

References

[1] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *NDSS*, 2019.

[2] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[3] ARM Limited. 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.

[4] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *TOCS*, 33(3):1–26, 2015.

[6] I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In *CRYPTO*, pages 131–146. Springer, 2000.

[7] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A. Sadeghi. Dr. sgx: automated and adjustable side-channel protection for sgx using data location randomization. In *ACSAC*, pages 788–800, 2019.

[8] F. Brasser, B. El Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *DAC*, pages 1–6. IEEE, 2015.

[9] F. Brasser, T. Frassetto, K. Riedhammer, A. Sadeghi, T. Schneider, and C. Weinert. Voiceguard: Secure and private speech processing. In *Interspeech*, pages 1303–1307, 2018.

[10] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *WOOT*, 2017.

[12] C. Canella, D. Genkin, L. Giner, D. Gruss, et al. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, pages 769–784, 2019.

[13] S. Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *ASPLOS*, volume 13, pages 253–264, 2013.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *EuroS&P*, pages 142–157. IEEE, 2019.

[15] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Asia CCS*, pages 7–18. ACM, 2017.

[16] H. D. Chirammal, P. Mukhedkar, and A. Vettathu. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.

[17] D. Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[18] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *USENIX ATC*, pages 565–578, 2016.

[19] K. Choi, K. Toh, and H. Byun. Realtime training on mobile devices for face recognition applications. *Pattern recognition*, 44(2):386–400, 2011.

[20] J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar. Seca: security-enhanced communication architecture. In *CASES*, pages 78–89. ACM, 2005.

[21] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2019.

[22] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[23] P. Cotret, J. Crenne, G. Gogniat, and J. Diguet. Bus-based mpsoc security through communication protection: A latency-efficient alternative. In *FCCM*, pages 200–207. IEEE, 2012.

[24] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, pages 463–478, 2013.

[25] G. Dessouky, T. Frassetto, and A. Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security*, 2020.

[26] EMBC. Coremark. https://www.eembc.org/coremark/, 2019.

[27] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*, pages 287–305. ACM, 2017.

[28] RISC-V Foundation. The risc-v instruction set manual, volume ii: Privileged architecture. https://riscv.org/specifications/privileged-isa/, 2019.

[29] RISC-V Foundation. Risc-v proxy kernel and boot loader. https://github.com/riscv/riscv-pk, 2019.

[30] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi. Imix: In-process memory isolation extension. In *USENIX Security*, pages 83–97, 2018.

[31] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *USENIX Security*, pages 955–972, 2018.

[32] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *MobiSys*, pages 488–501. ACM, 2017.

[33] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing arm trustzone. In *USENIX Security)*, 2017.

[34] Advanced Micro Devices Inc. Amd64 architecture programmer's manual volume 2: System programming. https://www.amd.com/system/files/TechDocs/24593.pdf, 2019.

[35] Intel. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[36] G. Irazoqui, T. Eisenbarth, and B. Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing and its application to aes. In *S&P*, pages 591–604. IEEE, 2015.

[37] N. P. Jouppi, C. Young, N. Patil, and D. Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, 2018.

[38] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[39] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *DAC*, page 72. ACM, 2016.

[40] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[41] C. King. stress-ng. https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html, 2019.

[42] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *MICRO*, pages 974–987. IEEE, 2018.

[43] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[44] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, et al. sel4: Formal verification of an os kernel. In *SOSP*, pages 207–220. ACM, 2009.

[45] P. Kocher, J. Horn, A. Fogh, D. Genkin, et al. Spectre attacks: Exploiting speculative execution. In *S&P*, pages 1–19. IEEE, 2019.

[46] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.

[47] P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *EuroSys*, page 10. ACM, 2014.

[48] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović. Keystone: A framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[49] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *S&P*, pages 19–33. IEEE, 2014.

[50] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[51] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *USENIX Security*, pages 1257–1272, 2019.

[52] LibTom. Libtomcrypt. https://www.libtom.net/LibTomCrypt/, 2019.

[53] ARM Limited. Trusted board boot requirements client (tbbr-client) armv8-a. https://static.docs.arm.com/den0006/d/DEN0006D_Trusted_Board_Boot_Requirements.pdf?_ga=2.193628069.980937939.1583698138-225494643.1545056698, 2018.

[54] ARM Limited. Amba® axi and ace protocol specification. https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf, 2019.

[55] Arm Limited. Arm® architecture reference manual. https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf, 2019.

[56] ARM Limited. Arm platform security architecture trusted boot and firmware update. https://pages.arm.com/rs/312-SAX-488/images/DEN0072-PSA_TBFU_1.0-bet1.pdf, 2019.

[57] Linaro. Op-tee. https://www.op-tee.org/.

[58] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, pages 406–418. IEEE, 2016.

[59] F. Liu and R. B. Lee. Random fill cache architecture. In *MICRO*, pages 203–215. IEEE, 2014.

[60] F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *MICRO*, 36(5):8–16, 2016.

[61] John M. Intel software guard extensions remote attestation end-to-end example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example, 2018.

[62] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[63] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *NDSS*, 2019.

[64] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*. ACM, 2013.

[65] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. Severed: Subverting amd's virtual machine encryption. In *EuroSec*. ACM, 2018.

[66] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, 2013.

[67] NVIDIA. Developing a linux kernel module using gpudirect rdma. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2019.

[68] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.

[69] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *USENIX ATC*, 2018.

[70] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *RSA Conference*, 2006.

[71] Orson P. ed25519. https://github.com/orlp/ed25519, 2019.

[72] R. D. Pietro, F. Lombardi, and A. Villani. Cuda leaks: a detailed hack for cuda and a (partial) fix. *TECS*, 15(1):15, 2016.

[73] M. Portnoy. *Virtualization essentials*, volume 19. John Wiley & Sons, 2012.

[74] M. K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787. IEEE, 2018.

[75] RV-8. Rv8-bench. https://github.com/rv8-io/rv8-bench, 2019.

[76] F. L. Sang, V. Nicomette, and Y. Deswarte. I/o attacks in intel pc-based architectures and countermeasures. In *SysSec Workshop*, pages 19–26. IEEE, 2011.

[77] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *USENIX Security*, pages 1357–1374, 2017.

[78] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019.

[79] M. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[80] SiFive. Sifive tilelink specification. https://sifive.cdn.prismic.io/sifive%2F57f93ecf-2c42-46f7-9818-bcdd7d39400a_tilelink-spec-1.7.1.pdf, 2018.

[81] SiFive. Sifive block inclusive cache. https://github.com/sifive/block-inclusivecache-sifive, 2019.

[82] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *S&P*, pages 1–17. IEEE, 2016.

[83] M. Sonka, V. Hlavac, and R. Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[84] D. Steinkraus, I. Buck, and P. Simard. Using gpus for machine learning algorithms. In *ICDAR*, pages 1115–1120. IEEE, 2005.

[85] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *DSN*, 2015.

[86] A. Tang, S. Sethumadhavan, and S. Stolfo. Clkscrew: exposing the perils of security-oblivious energy management. In *USENIX Security*, pages 1057–1074, 2017.

[87] C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *USENIX ATC*, pages 645–658, 2017.

[88] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *USENIX Security*, pages 1221–1238, 2019.

[89] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[90] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[91] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *CCS*, pages 178–195. ACM, 2018.

[92] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017.

[93] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. *S&P*, 2019.

[94] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020.

[95] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. Codoms: Protecting software with code-centric memory domains. In *ISCA*, pages 469–480. IEEE, 2014.

[96] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *USENIX OSDI 18*, pages 681–696, 2018.

[97] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *DAC*, pages 1–6. ACM, 2016.

[98] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A. Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[99] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. Scattercache: thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019.

[100] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *ISCA*, pages 457–468. IEEE, 2014.

[101] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, pages 640–656. IEEE, 2015.

[102] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[103] Google Projekt Zero. Trust issues: Exploiting trustzone tees. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017.

[104] Google Projekt Zero. Cve-2018-17182. https://bugs.chromium.org/p/project-zero/issues/detail?id=1664, 2018.

[105] Google Projekt Zero. Xnu: copy-on-write behavior bypass via mount of user-owned filesystem image. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2018.

[106] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *CCS*, pages 1723–1740. ACM, 2019.

[107] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu. Vulnerable gpu memory management: towards recovering raw data from gpu. *Proceedings on Privacy Enhancing Technologies*, 2017(2):57–73, 2017.

C

POSE: Practical Off-chain Smart Contract Execution (NDSS'23)

[68] Tommaso Frassetto, Patrick Jauernig, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi. POSE: Practical Off-chain Smart Contract Execution. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 24-27, 2023*, 2023. CORE Rank A*. Section 4.1.

POSE: **P**ractical **O**ff-chain **S**mart Contract **E**xecution

Tommaso Frassetto*, Patrick Jauernig*, David Koisser*, David Kretzler†,
Benjamin Schlosser†, Sebastian Faust† and Ahmad-Reza Sadeghi*
Technical University of Darmstadt, Germany
*first.last@trust.tu-darmstadt.de
†first.last@tu-darmstadt.de

***Abstract*—Smart contracts enable users to execute payments depending on complex program logic. Ethereum is the most notable example of a blockchain that supports smart contracts leveraged for countless applications including games, auctions and financial products. Unfortunately, the traditional method of running contract code *on-chain* is very expensive, for instance, on the Ethereum platform, fees have dramatically increased, rendering the system unsuitable for complex applications. A prominent solution to address this problem is to execute code *off-chain* and only use the blockchain as a trust anchor. While there has been significant progress in developing off-chain systems over the last years, current off-chain solutions suffer from various drawbacks including costly blockchain interactions, lack of data privacy, huge capital costs from locked collateral, or supporting only a restricted set of applications.**

In this paper, we present *POSE*—a practical off-chain protocol for smart contracts that addresses the aforementioned shortcomings of existing solutions. *POSE* leverages a pool of Trusted Execution Environments (TEEs) to execute the computation efficiently and to swiftly recover from accidental or malicious failures. We show that *POSE* provides strong security guarantees even if a large subset of parties is corrupted. We evaluate our proof-of-concept implementation with respect to its efficiency and effectiveness.

I. Introduction

More than a decade ago, Bitcoin [47] introduced the idea of a decentralized cryptocurrency, marking the advent of the blockchain era. Since then, blockchain technologies have rapidly evolved and a plethora of innovations emerged with the aim to replace centralized platform providers by distributed systems. One particularly important application of blockchains concerns so-called *smart contracts*, complex transactions executing payments that depend on programs deployed to the blockchain. The first and most popular blockchain platform that supported complex smart contracts is Ethereum [58]. However, Ethereum still falls short of the decentralized "world computer" that was envisioned by the community [51]. For example, contracts are replicated among a large group of miners, thereby severely limiting scalability and leading to high costs. As a result, most contracts used in practice in the Ethereum ecosystem are very simple: 80% of popular contracts consist of less than 211 instructions, and almost half of the most active contracts are simple token managers [49]. More recently proposed computing platforms in permissionless decentralized settings (e.g., [1], [34]) suffer from similar scalability limitations.

In recent years, numerous solutions have been proposed to address these shortcomings of blockchains, one of the most promising being so-called *off-chain execution* systems. These protocols move the majority of transactions off-chain, thereby minimizing the costly interactions with the blockchain. A large body of work has explored various types of off-chain solutions including most prominently state-channels [46], [26], [22], Plasma [52], [37] and Rollups [48], [5], which are actively investigated by the Ethereum research community. Other schemes use execution agents that need to agree with each other [60], [59], rely on incentive mechanisms [36], [57], or leverage Trusted Execution Environments (TEEs) [20], [25]. A core challenge that arises while designing off-chain execution protocols is to handle the possibility of parties who stop responding, either maliciously or accidentally. Without countermeasures, this may cause the contract execution to stop unexpectedly, which violates the *liveness* property. Despite major progress towards achieving liveness in a off-chain setting, current solutions come with at least one of these limitations: i participating parties need to lock large amounts of collateral; ii costly blockchain interactions are required at every step of the process or at regular intervals; and finally iii the set of participants and the lifetime need to be known beforehand, which limits the set of applications supported by the system. Additionally, existing solutions often iv do not support keeping the contract state confidential, which is required, e.g., for eBay-style proxy auctions [9] and games such as poker. We refer the reader to Table II for an overview on related work and to Section X for a detailed discussion.

Addressing all of these limitations in one solution while guaranteeing liveness is highly challenging. Currently, there are two ways to address the risks of unresponsive parties. The first approach is to require collateral, i.e., parties have to block large amounts of money, which is used to disincentivize malicious behavior and to compensate parties in case of premature termination (cf. i). Since the amount of collateral depends on the number of participants and the amount of money in the contract, both must be fixed for the whole lifetime of the contract. To ensure payout of the collateral, the lifetime of the contract must be fixed as well (cf. iii). The second approach is to store contract state on the blockchain to enable other parties to resume execution. However, this is both expensive and leads to long waiting times due to frequent synchronization with the blockchain (cf. ii). Further, if the contract state needs to be confidential, and hence, is not publicly verifiable,

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23118
www.ndss-symposium.org

verifying the correctness of the contract execution is harder (cf. iv). Realizing a system tackling all these challenges in a holistic way could pave the way towards the envisioned "world computer". We will further elaborate on the specific challenges in Section III.

Our goals and contributions: We present *POSE*, a novel off-chain execution framework for smart contracts in permissionless blockchains that overcomes these challenges, while achieving correctness and strong liveness guarantees. In *POSE*, each smart contract runs on its own subset of TEEs randomly selected from all TEEs registered to the network. One of the selected TEEs is responsible for the execution of a smart contract.

However, as the system hosting the executing TEE may be malicious (e.g., the TEE could simply be powered off during contract execution), our protocol faces the challenge of dealing with malicious operator tampering, withholding and replaying messages to/from the TEE. Hence, the TEE sends state updates to the other selected TEEs, such that they can replace the executing TEE if required. This makes *POSE* the first off-chain execution protocol with strong liveness guarantees. In particular, liveness is guaranteed as long as at least one TEE in the execution pool is responsive. Due to this liveness guarantee, there is no inherent need for a large collateral in *POSE* (cf. i). The state remains confidential, which allows *POSE* to have private state (cf. iv). Furthermore, *POSE* allows participants to change their stake in the contract at any time. Thus, *POSE* supports contracts without an a-priori fixed lifetime and enables the set of participants to be dynamic (cf. iii). Above all, *POSE* executes smart contracts quickly and efficiently without any blockchain interactions in the optimistic case (cf. ii).

This enables the execution of highly complex smart contracts and supports emerging applications to be run on the blockchain, such as federated machine learning. Thus, *POSE* improves the state of the art significantly in terms of security guarantees and smart contract features. To summarize, we list our main contributions below:

- We introduce *POSE*, a fast and efficient off-chain smart contract execution protocol. It provides strong guarantees without relying on blockchain interactions during optimistic execution, and does not require large collaterals. Moreover, it supports contracts with an arbitrary contract lifetime and a dynamic set of users. An additional unique feature of *POSE* is that it allows for confidential state execution.
- We provide a security analysis in a strong adversarial model. We consider an adversary which may deviate arbitrarily from the protocol description. We show that *POSE* achieves correctness and state privacy as well as strong liveness guarantees under static corruption, even in a network with a large share of corrupted parties.
- To illustrate the feasibility of our scheme, we implement a prototype of *POSE* using ARM TrustZone as the TEE and evaluated it on practical smart contracts, including one that can merge models for federated machine learning in 238ms per aggregation.

II. Adversary Model

The goal of *POSE* is to allow a set of users to run a complex smart contract on a number of TEE-enabled systems. Note, that *POSE* is TEE-agnostic and can be instantiated on any TEE architecture adhering to our assumptions, similar to, e.g., FastKitten [25]. In order to model the behavior and the capabilities of every participant of the system, we make the following assumptions:

A1: We assume the TEE to protect the enclave program, in line with other TEE-assisted blockchain proposals [63], [25], [20], [17], [64], [43]. Specifically:
A1.1: We assume the TEE to provide integrity and confidentiality guarantees. This means that the TEE ensures that the enclave program runs correctly, is not leaking any data, and is not tampering with other enclaves. While our proof of concept is based on TrustZone, our design does not depend on any specific TEE. In practice, the security of a TEE is not always flawless, especially regarding information leaks. However, plenty of mitigations exist for the respective commercial TEEs; hence, we consider the problem of information leakage from any specific TEE, as well as TEE-specific vulnerabilities in security services, orthogonal to the scope of this paper. We discuss some mitigations to side-channel attacks to TrustZone, as well as the possible grave consequences of a compromised or leaking TEE for the executed smart contract, in Section VII-B.
A1.2: We further assume the adversaries to be unable to exploit memory corruption vulnerabilities in the enclave program. This could be ensured using a number of different approaches, e.g., by using memory-safe languages, by deploying a run-time defense like CFI [11], or by proving the correctness of the enclave program using formal methods. The existence of these defenses can be proven through remote attestation (cf. A3).
A2: We assume the TEE to provide a good source of randomness to all its enclaves and to have access to a relative clock according to the GlobalPlatform TEE specification [32].
A3: We assume the TEE to support *secure remote attestation*, i.e., to be able to provide unforgeable cryptographic proof that a specific program is running inside of a genuine, authentic enclave. Further, we assume the attestation primitive to allow differentiation of two enclaves running the same code under the same data. Note that today's industrial TEEs support remote attestation [3], [6], [8], [35], [56].
A4: We assume the TEE operators, i.e., the persons or organizations owning the TEE-enabled machines, to have full control over those machines, including root access and control over the network. The operators can, for instance, provide wrong data to an enclave, delay the transmission of messages to it, or drop messages completely. The operators can also completely disconnect an enclave from the network or (equivalently) power off the machine containing it. However, as stated in A1.1, the operators cannot leak data from any enclave or influence its computation in any way besides by sending (potentially malicious) messages to it through the official software interfaces.
A5: We assume static corruption by the adversary. More precisely, a fixed fraction of all operators is corrupted while an arbitrary number of users can be malicious (including the case where they all are). We model each of the malicious parties as *byzantine adversaries*, i.e., they can behave in arbitrary ways.
A6: We assume the blockchain used by the parties to satisfy

the following standard security properties: common prefix (ignoring the last γ blocks, honest miners have an identical chain prefix), chain quality (blockchain of honest miner contains significant fraction of blocks created by honest miners), and chain growth (new blocks are added continuously). These properties imply that valid transactions are included in one of the next α blocks and that no valid blockchain fork of length at least γ can grow with the same block creation rate as the main chain. We deem protection against network attacks (e.g., network partition attacks), which violate these standard properties, orthogonal to our work.

III. Design

POSE is a novel off-chain protocol for highly efficient smart contract execution, while providing strong correctness, privacy, and liveness guarantees. To achieve this, *POSE* leverages the integrity and confidentiality guarantees of TEEs to speed up contract execution and make significantly more complex contracts practical[1]. This is in contrast to executing contracts on-chain, where computation and verification is distributed over many parties during the mining process. *POSE* supports contracts with arbitrary lifetime and number of users, which includes complex applications like the well-known CryptoKitties [2]. We elaborate more on interaction between contracts in Appendix B. Our protocol involves users, operators and a single on-chain smart contract. *Users* aim to interact with smart contracts by providing inputs and obtaining outputs in return. *Operators* own and manage the TEE-enabled systems and contribute computing power to the *POSE* network by creating protected execution units, called *enclaves*, using their TEEs. These enclaves perform the actual state transitions triggered by users. A simple on-chain smart contract, which we call *manager*, is used to manage the off-chain enclave execution units. In the optimistic case, when all parties behave honestly, *POSE* requires only on-chain transactions for the creation of a *POSE* contract as well as the locking and unlocking of user funds. The smart contract execution itself is done without any on-chain transactions.

A. Architecture Overview

Figure 1 illustrates the high-level working of *POSE*. Before contract creation, there is already a set of enclaves that are registered with the on-chain manager contract. The registration process is explained in detail in Section V-E1. To create a *POSE* contract, a user will initialize a contract creation with the manager (Step 1), which includes a chosen enclave—out of the registered set—to execute the off-chain contract creation. In Step 2, the chosen *creator* enclave will setup the *execution pool* for the given smart contract. In Figure 1, the pool size is set to three; thus, the *creator* enclave will randomly select three enclaves from the set of all enclaves registered in the system (Step 3). In Step 4, the *creator* enclave will submit the finalized contract information to the manager. This includes the composition of the execution pool, i.e., a selected *executor* enclave, which is responsible for executing the *POSE* contract, as well as the *watchdogs*, ensuring availability. We elaborate on this in-depth in Section V-E2. In Step 5, another user can

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

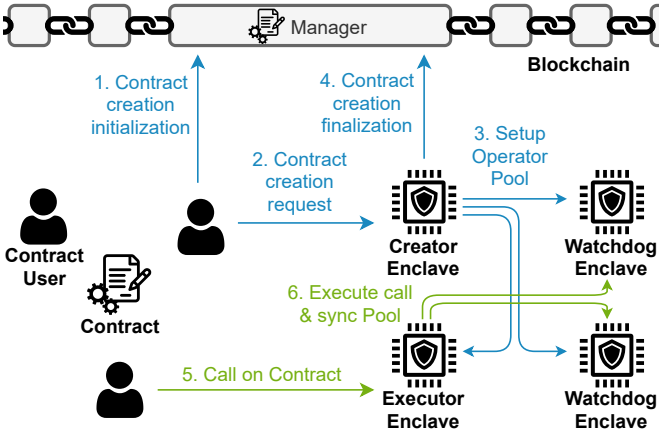


Fig. 1. Exemplary overview how *POSE* contracts are created (in blue) and executed (in green).

now call the new contract by directly contacting the executor. Finally, for Step 6, the executor will execute the user's contract call and distribute the resulting state to the watchdog enclaves, which confirm the state update. See Section V-E3 for a detailed specification of the execution protocol. If one of the enclaves stops participating (e.g., due to a crash), the dependent parties can challenge the enclave on the blockchain (see Section V-E4). The dependent party can either be the user awaiting response from the executor or the executor waiting for the watchdogs' confirmation. For example, if the executor stops executing the contract, the executor is challenged by the user. A timely response constitutes a successful state transition as requested by the user. Otherwise, if the current executor does not respond, one of the watchdogs will fill in as the new executor. This makes *POSE* highly available, as long as at least one watchdog enclave is dependable; thus, avoiding the need for collateral to incentivize correct behavior. Further, *POSE* supports private state, as the state is only securely shared with other enclaves.

B. Design Challenges

We encountered a number of challenges while designing *POSE*. We briefly discuss them below.

Protection Against Malicious Operators. *POSE*'s creator, executor, and watchdogs are protected in isolated enclaves running within the system, which is itself still under control of a potentially malicious operator. Hence, operators can provide arbitrary inputs, modify honest users' messages, execute replay attacks, and withhold incoming messages. Moreover, the system and its TEE (i.e., enclaves) can be turned off completely by its operator. In order to protect honest users from malicious operators, we incorporate several security mechanisms. While malicious inputs and modification of honest users' messages can easily be prevented using standard measures like a secure signature scheme, preventing withholding of messages is more challenging. One particular reason is that for unreceived messages, an enclave cannot differentiate between unsent and stalled messages by the operator. Hence, we incorporate an on-chain challenge-response procedure, which provides evidence about the execution request and the existence of a response to the enclave.

Achieving Strong Liveness Guarantees. We enable dependent parties to challenge unresponsive operators via the blockchain. The challenged operators either provide valid responses over the blockchain that dependent parties can use to finalize the state transition, or they are dropped from the execution pool. In case an executor operator has been dropped, we use the execution pool to resume the execution; this requires state updates to be distributed to all watchdogs. With at least one honest operator in the execution pool, the pool will produce a valid state transition. Our protocol tolerates a fixed fraction of malicious operators as stated in our adversary model (cf. Section II). By selecting the pool members randomly, we guarantee with high probability that at least one enclave—controlled by an honest operator—is part of the execution pool. We show in Section VII-A that our protocol achieves strong liveness guarantees.

Synchronization with the Blockchain. Some of the actions taken by an enclave depend on blockchain data, e.g., deposits made by clients. Hence, it is crucial to ensure that the blockchain data available to an enclave is consistent and synchronized with the main chain. As an enclave does not necessarily have direct access to the (blockchain) network, it has to rely on the blockchain data provided by the operator. However, the operator can tamper with the blockchain data and, e.g., withhold blocks for a certain time. Thus, a major challenge is designing a synchronization mechanism that (i) imposes an upper bound on the time an enclave may lag behind the main chain, (ii) prevents an operator from isolating an enclave onto a fake side-chain, and (iii) ensures correctness and completeness of the blockchain data provided to the enclave, without (iv) requiring the enclave to validate or store the entire blockchain. We present our synchronization mechanism addressing these challenges in Section V-D.

Reducing Blockchain Interactions. Our system aims to minimize the necessary blockchain interactions to avoid expensive on-chain computations. In the optimistic scenario, the only on-chain transactions necessary are the contract creation and the transfer of coins. The transfer transactions can also be bundled to further reduce blockchain interactions. Note that the virtualization paradigm known from state channels [26] can be applied to our system. This enables parties to install virtual smart contracts within existing smart contracts, and hence, without any on-chain interactions at all. In the pessimistic scenario, i.e., if operators fail to provide valid responses, they have to be challenged, which requires additional blockchain interactions.

Support of Private State. To support private state of randomized contracts, careful design is required to avoid leakage. While the confidentiality guarantees of TEEs prevent any data leakage during contract execution, our protocol needs to ensure that an adversary cannot learn any information except the output of a successful execution. In particular, in a system where the contract state is distributed between several parties, we need to prevent the adversary from performing an execution on one enclave, learning the result, and exploiting this knowledge when rolling back to an old state with another enclave. This is due to the fact that a re-execution may use different randomness or different inputs resulting in a different output. We prevent these attacks by outputting state updates to the users only if all pool members are aware of the new state. Moreover, by solving the challenge of synchronization between enclaves and the blockchain, we prevent an adversary from providing a fake chain to the enclave, in which honest operators are kicked from the execution pool. Such a fake chain would allow an attacker to perform a parallel execution. While results of the parallel (fake) execution cannot affect the real execution, they can prematurely leak private data, e.g. the winner in a private auction.

IV. Definitions & Notations

In the following, we introduce the cryptography primitives, definition, and notations used in the *POSE* protocol.

Cryptographic Primitives. Our protocol utilizes a public key encryption scheme $(GenPK, Enc, Dec)$, a signature scheme $(GenSig, Sign, Verify)$, and a secure hash function $H(\cdot)$. All messages sent within our protocol are signed by the sending party. We denote a message m signed by party P as $(m; P)$. The verification algorithm $Verify(m')$ takes as input a signed message $m' := (m; P)$ and outputs `ok` if the signature of P on m is valid and `bad` otherwise. We identify parties by their public keys and abuse notation by using P and P's public key pk_P interchangeably. This can be seen as a direct mapping from the identity of a party to the corresponding public key.

TEE. We comprise the hardware and software components required to create confidential and integrity-protected execution environments under the term TEE. An operator can instruct her TEE to create new *enclaves*, i.e., new execution environments running a specified program. We follow the approach of Pass et al. [50] to model the TEE functionality. We briefly describe the operations provided by the ideal functionality formally specified in [50, Fig. 1]. A TEE provides a $TEE.install(prog)$ operation which creates a new enclave running the program $prog$. The operation returns an enclave id eid. An enclave with id eid can be executed multiple times using the $TEE.resume(eid, inp)$ operation. It executes $prog$ of eid on input inp and updates the internal state. This means in particular that the state is stored across invocations. The $resume$ operation returns the output out of the program. We slightly deviate from Pass et al. [50] and include an attestation mechanism provided by a TEE that generates an attestation quote ρ over $(eid, prog)$. ρ can be verified by using method $VerifyQuote(\rho)$. We consider only one instance $\mathcal{E}$ running the *POSE* program per TEE. Therefore, we simplify the notation and write $\mathcal{E}(inp)$ for $TEE.resume(eid, inp)$.

Blockchain. We denote the blockchain by BC and the average block time by τ. A block is considered final if it has at least γ confirmation blocks. Throughout the protocol description in Section V-E, enclaves consider only transactions included in final blocks. Finally, we define that any smart contact deployed to the blockchain is able to access the current timestamp using the method $\mathsf{BC}.now$ and the hash of the most recent 265 blocks [7] using the method $\mathsf{BC}.bh(i)$ where i is the number of the accessed block. These features are available on Ethereum.

V. The *POSE* Protocol

The *POSE* protocol considers four different roles: a manager smart contract deployed to the blockchain, operators that run TEEs, enclaves that are installed within TEEs, and users

that create and interact with *POSE* contracts. In the following, we will shortly elaborate on the on-chain smart contract and the program executed by the enclaves, explain the *POSE* protocol, and finally explain further security mechanisms that are omitted in the protocol description.

A. Manager

We utilize an on-chain smart contract in order to manage the *POSE* system's on-chain interactions. We call this smart contract *manager* and denote it by M. On the one hand, M keeps track of all registered *POSE* enclaves. This enables the setup of an execution pool whenever an off-chain smart contract instance is created. On the other hand, it serves as a registry of all *POSE* contract instances. M stores parameters about each contract to determine the instance's status. We denote the tuple describing a contract with identifier id as M^{id}. In particular, the manager stores the creator enclave ($creator$), a hash of the program code ($codeHash$), the set of enclaves forming the execution pool ($pool$), a total amount of locked coins ($balance$), and a counter of withdrawals ($payouts$). We set the field $creator$ to $\perp$ after the creation process has been completed to identify that a contract is ready to be executed. Moreover, for both executor and watchdog challenges, the contract allocates storage for a tuple containing the challenge message ($c1Msg$ resp. $c2Msg$), responses ($c1Res$ resp. $c2Res$), and the timestamp of the challenge submission ($c1Time$ resp. $c2Time$). A non-empty field $c1Time$ resp. $c2Time$ signals that there is a running challenge.

Every *POSE* enclave maintains a local version of the manager state extracted from the blockchain data it receives from the operator when being executed. This enables all enclaves to be aware of on-chain events, e.g., ongoing challenges.

B. POSE Program

All enclaves registered within the system run the *POSE* program that enforces correct execution and creation of *POSE* contracts. In practice, the *POSE* program's source code will be publicly available, e.g., in a public repository, so that the community can audit it. Our protocol ensures that all registered enclaves run this code using remote attestation (cf. Section V-E1: Enclave registration). We present methods required for the execution protocol in Program 1 and defer methods for the contract creation to the full version of this paper [31].

Whenever an enclave is invoked, it synchronizes itself with the blockchain network and receives the relevant blockchain data in a reliable way (cf. Section V-D). This way, the POSE program has access to the current state of the manager. In order to support arbitrary contracts, we define a common interface in Section V-C that is used by the POSE program to invoke contracts.

Enclaves running the *POSE* program only accept signed messages as input. The public keys of pool members for signature verification are derived from the synchronized blockchain data. According to our adversary model (cf. Section II), the adversary cannot read or tamper messages originating from honest users or the enclave itself. Further, the contracts themselves keep track of already received execution requests and do not perform state transitions for duplicated requests.

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

Upon invocation with input blockchain data BC, store BC.
Upon receiving $m := (\texttt{execute}, id, r, move; U)$, 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 $(\texttt{ok}, id, state, h^{id}; T)$.

(cf. Section V-C). This prevents replay attacks against both, executive and watchdog enclaves.

C. POSE Contracts

Although our system supports the execution of arbitrary smart contracts, the contracts need to implement a specific interface (cf. Program 2). This allows any *POSE* enclave to trigger the execution without knowing details about the smart contract functionality. Upon an execution request from some user, the *POSE* enclave provides the user's identity U, blockchain data BC, the description of the user's request, $move$, and the request hash, h, to the smart contract's method $nextState$. The smart contract first processes the relevant blockchain data and marks the current length of the blockchain as processed. This feature is mainly used to enable smart contracts to deal with money, i.e., to detect on-chain deposits and withdrawals. We elaborate on the processing of blockchain data in Section V-D, and on the money mechanism of the *POSE* system in Appendix E. Note that double spending within a contract is prevented due to sequential processing of any execution request, and double spending of on-chain payouts is prevented by the mechanism explained in Appendix E. After the blockchain data is processed, $nextState$ executes the move requested by the user and updates the state accordingly. Method $update$ takes state new and hash h (for preventing replay attacks) as input and sets new as the contract state. This includes the length of the blockchain that is marked as processed. Further, the smart contract provides method $getState$. If called with $flag = all$, it returns the whole smart contract state. Otherwise, if called with $flag = pub$, it returns only the public state. In order to prevent replay attacks, each smart contract maintains a list with the hashes of already received execution requests, Rec. In case of duplicated requests, i.e., $h \in Rec$, both the $nextState$ method and the $update$ method, do not perform any state transition. Instead, they interpret the request as a dummy move that has no effect on the state. If executed successfully, the $nextState$ method

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

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

adds the executed request to Rec, i.e., $Rec = Rec \cup \{h\}$. As Rec is part of the state, it is updated by the $update$ method as well. While it might seem counter intuitive to overwrite the list of received requests, this feature is required to ensure that all enclaves are aware of the same transition history; even if an executor distributes a state update to just a subset of watchdogs before getting kicked [2].

We consider the initial state of a smart contract to be hard-coded into the smart contract description. If an enclave creates a new smart contract instance, the initial state is automatically initialized. A contract state additionally contains a variable to store the highest block number of the already processed blockchain data. This variable is used to detect which transactions of received blockchain data have already been handled.

D. Synchronization

As some of the actions taken by an enclave depend on blockchain data, e.g., deposits to the contract, it is crucial to ensure that the blockchain state available to a registered enclave $\mathcal{E}$ is consistent and synchronized with the main chain. In particular, blocks that are considered final by some party, will eventually be considered final by all parties. We design a synchronization mechanism that allows $\mathcal{E}$ to synchronize itself without having to validate whole blocks. Note that $\mathcal{E}$ has access to a relative time source according to our adversary model (see Section II).

Upon initialization, $\mathcal{E}$ receives a chain of block headers BCH of length $\gamma + 1$. Note that the first block p of BCH can be considered final since it has γ confirmation blocks. First, $\mathcal{E}$ checks that BCH is consistent in itself and sets its own clock to be the one of the latest block's timestamp. Second, $\mathcal{E}$ signs block p as blockchain evidence that needs to be provided to the manager. The registration mechanism (cf. Section V-E1) uses this evidence to ensure that $\mathcal{E}$ has been initialized with a valid sub-chain of the main-chain up to block p. Further, the registration mechanism checks that p is at most τ^{on}_{slack} blocks behind the current one; τ^{on}_{slack} 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^{off}_{slack} an enclave may lag behind; τ^{off}_{slack} additionally considers potential block variance and the fact that miners have some margin to set timestamps. In the following, we call τ^{off}_{slack} *slack* [3]. Clients that want a contract execution to capture on-chain effects, e.g., deposits, wait until the enclave considers the corresponding block as final, even when being at slack.

Once successfully initialized, $\mathcal{E}$ synchronizes itself with the blockchain. Whenever a registered enclave is executed throughout the protocol, it receives the sub-chain of block headers BCH' that have been mined since the last execution. $\mathcal{E}$ checks that BCH' is a valid successor of BCH where blocks in BCH that have not been final may change. Further, $\mathcal{E}$ checks that the latest block in BCH' is at most $\tau_{variance}$ behind the own clock; $\tau_{variance}$ captures the variance in the block creation time and the fact that miners have some margin to set timestamps. When receiving a block that is before the own clock, the clock is adjusted.

Finally, we need to prevent an operator from isolating its enclave by setting up a valid sidechain with manipulated timestamps. To this end, we require the operators to periodically provide new blocks to $\mathcal{E}$ even if $\mathcal{E}$ does not need to take any action. In particular, we require that the operator provides at least L blocks within time τ_p where τ_p accounts for potential block time variances. The system is secure as long as the attacker cannot mine L blocks within time τ_p while the honest miners can. Hence, the selection of τ_p and L has some implications on the fraction of adversarial computing power that can be tolerated by the system. Since 2018, an interval of 50 (100, 200, 300) blocks took at most 33 (28, 26, 25) seconds per block [10], which might all be reasonable choices for L and $\frac{\tau_p}{L}$. As the average block time is around 13 seconds [4], the adversary gets $2-3$ times more time to mine the blocks of its sidechain. This means that the system can tolerate adversarial fractions from a third (when instantiated with $L = 300$ and $\tau_p = 25 \cdot L$) to a forth (when instantiated with 50 and $33 \cdot L$).

While the above techniques allow an enclave to synchronize itself, the enclave does not have access to the block data, yet. Instead of requiring enclaves to validate whole blocks, we require operators to filter the relevant transactions and provide them to the enclave while enabling the enclaves to check correctness and completeness of the received data itself. For the latter, we introduce $incrTxHash$, a hash maintained by the manager and all initialized enclaves that is based on all relevant transactions. Whenever the manager receives a relevant transaction tx, it updates $incrTxHash$, such that $incrTxHash_{i+1}$ is defined as

$$H(incrTxHash_i \mid\mid tx.data \mid\mid tx.sender \mid\mid tx.value)$$

where $tx.data$ is the raw data of tx, $tx.sender$ denotes the creator of tx, and $tx.value$ contains the amount of any deposits or withdrawals. Whenever enclaves are invoked with new blocks, operators additionally provide all relevant transactions. This way, enclaves can re-compute the new incremental hash and compare the result to the on-chain value of $incrTxHash$. In order to verify that the on-chain $incrTxHash$ is indeed part of the main chain, operators additionally provide a Merkle proof showing that $incrTxHash$ is part of the state tree. The proof can be validated using the state root, which is part of the block headers provided to the enclaves. This way, enclaves can ensure that operators have not omitted or manipulated any relevant transactions.

[2] 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.

[3] 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.

E. Protocol Description

In this section, we dive into a detailed description of our protocol. We present 1) enclave registration, 2) contract creation, 3) contract execution, and 4) the challenge-response parts of our protocol. The *POSE* program running inside the operators' enclaves is stated in Section V-B. For the sake of exposition, we extracted the validation steps performed by the manager on incoming messages into Program 3 in Appendix C. Further, we elaborate in Appendix E on the coin flow within the protocol.

1) Enclave Registration: Operator O controlling some TEE unit can contribute to the *POSE* system by instructing his TEE to create a new *POSE* enclave $\mathcal{E}_O$. The protected execution environment $\mathcal{E}_O$ needs to be initialized with the *POSE* program presented in Section V-B. During the creation of $\mathcal{E}_O$, an asymmetric key pair (pk_O, sk_O) is generated. The secret key sk_O is stored inside the enclave and hence is only accessible by the *POSE* program running in $\mathcal{E}_O$. The public key pk_O is returned as output to the operator. Furthermore, operator O uses the TEE to produce an attestation ρ_O stating that the freshly generated enclave $\mathcal{E}_O$ runs the *POSE* program and controls the secret key corresponding to pk_O.[4]

Finally, O sends the latest $\gamma{+}1$ block headers BCH together with the relevant blockchain data to the enclave which validates the consistency of the block headers and completeness of the blockchain data (cf. Section V-D) and returns a blockchain evidence ρ_O^{BC}, i.e., a signed tuple containing the blockhash and the number of the latest final block known to the enclave. After operator O created a new *POSE* enclave $\mathcal{E}_O$, O can register $\mathcal{E}_O$ by sending $m := (\texttt{register}, \mathcal{E}_O, \rho_O, \rho_O^{\mathsf{BC}}; O)$ 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., $\mathit{Verify}(m) = \texttt{ok}$, 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.

2) Contract Creation: The creation protocol is initiated by a user U who wants to install a new smart contract, with program code code, into the *POSE* system. We outline the protocol in the following and provide a full explanation and specification in the full version of this paper [31].

U picks an arbitrary registered enclave $\mathcal{E}_C$ and sends a creation initialization to M containing $H(\mathit{code})$ 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 V-E4).

3) Contract Execution: The execution protocol is initiated by a user U who wants to execute an existing smart contract, identified by id, with input move. The protocol is specified in Figure 2. Program 1 specifies the parts of the *POSE* program that are relevant for the contract execution.

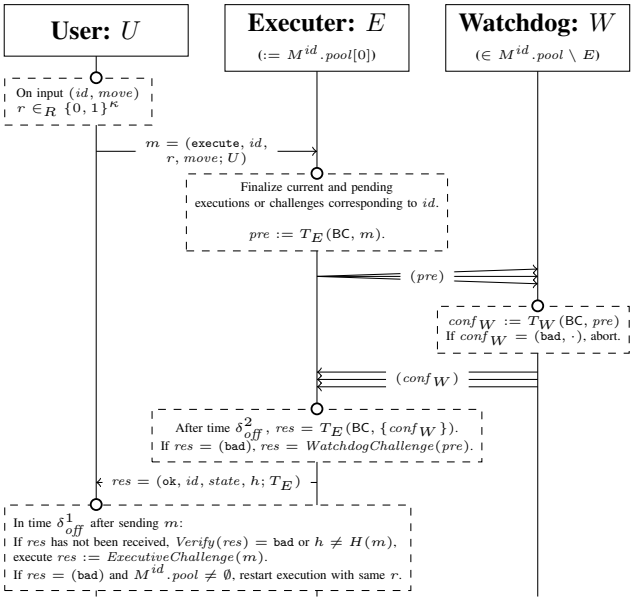


Fig. 2. Detailed execution protocol.

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'

[4] 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].

essentially contains both executions, $update$ and $update'$. In case $\mathcal{E}'_E$ has not received $update$ but the other watchdogs have, $\mathcal{E}'_E$ either propagates the update already known to the watchdogs, i.e., $update = update'$, or a concurrent one, i.e., $update \neq update'$. For the former, the watchdogs interpret the update as a dummy update without any effect as the corresponding execution request is already within their list of received request hashes (cf. Section V-C). For the latter, the update of the watchdogs is overwritten by the one of the executive enclave. As $update$ has been incomplete, and hence, produced no public output, it is safe to overwrite this update. To produce a public output for $update$, all pool enclaves including $\mathcal{E}'_E$ would have to confirm $update$.

Finally, U can just submit the previous execution request with the same random nonce r to $\mathcal{E}'_E$. In case the enclave has already seen this request, it is interpreted as empty dummy move which prevents a duplicated execution.

4) Challenge-Response: If any party does not receive a *timely* response to its messages during the off-chain execution, it challenges the receiver on-chain. Therefore, all operators need to monitor the blockchain for any on-chain challenges. We will elaborate on the timeouts $(\delta^{\dagger}_{\star})$, where $\dagger \in \{0,1\}$ and $\star \in \{off, on\}$, which define the notion of *timely* in Appendix D. In particular, we describe the relation between δ^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 V-E3. However, the operator prioritizes challenges over off-chain execution requests to avoid getting kicked. Additionally, if $\mathcal{E}_E$ already performed the state update and state propagation, the operator may use the already obtained result as response. Either way, if the operator sends a response message in time, the manager M checks the validity of the message and whether or not it matches the stored challenge. If all checks succeed, M stores the result and removes the

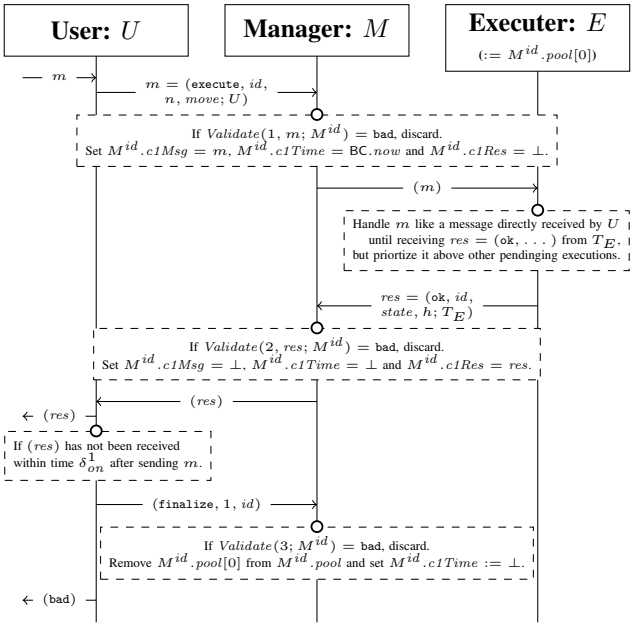


Fig. 3. Detailed executor challenge protocol.

challenge message. This finalizes the challenge procedure. If the operator does not send a valid response in time δ^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.

F. Security Remarks

To keep the protocol description compact, we omitted some security features from the specification, which we explain in this section.

Allowing unrestricted execution requests comes with the problem that malicious users can send requests whose execution takes a disproportional amount of time, e.g., due to infinite loops. If the execution time exceeded the boundaries defined by the on-chain timeouts, malicious users could exploit

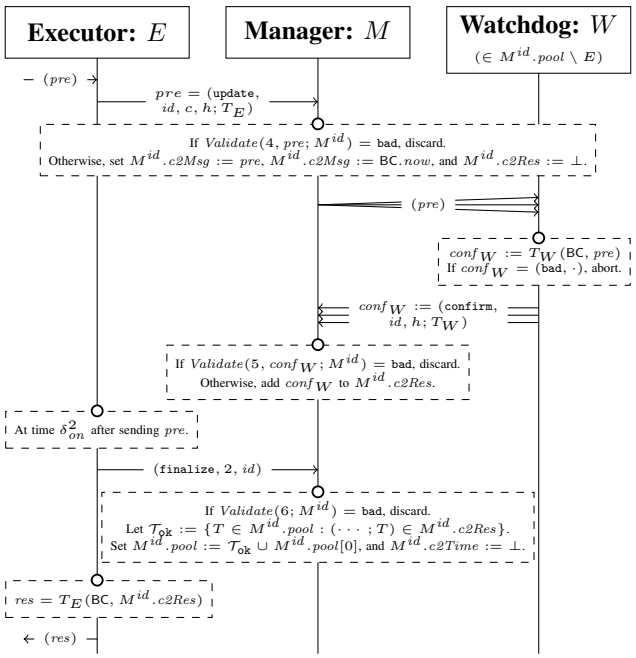


Fig. 4. Detailed watchdog challenge protocol.

this behavior to kick honest operators from an execution pool. This operator *denial of service* attack harms the liveness property of the system. In order to mitigate the vulnerability, we introduce an upper bound to the computation complexity of a single contract execution. Once the bound is reached, the executor enclave stops executing and reverts the state but still provides a valid output. The timeouts in the system are set such that an honest operator cannot be kicked from an execution pool even if an execution takes the maximum amount of computation. The same applies to update and creation requests, where failed creations return a *fail confirmation* that can be submitted to the manager instead of the creation confirmation. A fail confirmation triggers the manager to mark the contract as crashed. Note that the *POSE* system still supports the execution of arbitrary complex smart contracts as the timeouts and hence the upper bounds can be set arbitrarily high (cf. Appendix D). Additionally, all contracts of an operator are executed and challenged independently, and thus, contracts do not block each other.

While we have assumed that all operators run only one *POSE* enclave, multiple enclaves can be created in practice. This enables the opportunity of a *sybil attack*, where a malicious operator generates multiple *POSE* enclaves to increase its share in the system and hence harm the liveness property. This attack can be mitigated by forcing an operator to deposit funds at each enclave registration and which will be paid back to the operator only if she behaves honestly. We note that this deposit is independent of any contract and its parties. Now, such an attack is directly linked to financial loss. See Section VI for more discussions about incentives and fees.

In order to enhance *privacy*, neither users nor operators send inputs or respectively execution results in clear. Instead, users encrypt inputs using hybrid encryption based on the public key of the executor enclave. Additionally, users specify a symmetric key in their execution request, which is used to encrypt the result of the execution when sent back to the user. This way, inputs and results are private and cannot be eavesdropped by a malicious operator.

The term *griefing* denotes attacks where an adversary forces an honest party to interact with the blockchain in order to generate financial damage to this party. Especially when blockchain transactions require high fees, such attacks pose serious vulnerabilities. In regards to challenges within the *POSE* protocol, we mitigate the attack surface for griefing attacks by incorporating a mechanism in the manager that fairly splits the fees for challenge and response between the challenger and the challenged party. The same mechanism can be used for the contract creation process.

An adversary executing a *clogging* attack sends many transactions to the system to prevent honest users from issuing transactions. In the context of *POSE*, an off-chain clogging attack results in honest clients making an on-chain challenge to ensure that their requests will be processed. Hence, a successful clogging attack has to be performed on-chain. For the on-chain challenge, our system inherits the vulnerabilities of the underlying blockchain.

VI. Extensions

We simplified some protocol steps in order to make the protocol description more compact and easier to understand. We discuss the most important extensions and their benefits in this section.

Contract & Operator Lifecycle. A mechanism that releases enclaves from their execution duty can be integrated. This allows operators to voluntarily withdraw their enclaves from an execution pool. On the one hand, terminated contracts can be closed, which releases all pool enclaves from their execution duty. On the other hand, it enables to withdraw a single enclave and exchanging it by a randomly chosen replacement enclave. Additionally, a replacement strategy is also applicable to the scenarios in which enclaves are kicked. The latter extension reduces the chance of a contract crash, the event in which no more operator remains. We stress that these extensions can easily be achieved by adding the functionality to our *POSE* program and the manager. In case a contract is idle for a long time, an extension may be implemented that allows operators to *hibernate* their respective enclave. The enclave state can be stored on disk by encrypting it with a key that is kept alive in the hibernating enclave; thus, only requiring minimal overhead in memory. The *POSE* program ensures freshness by synchronizing with the blockchain; thus, preventing rollback attacks.

Incentives. Although *POSE* provides security not only against rational but also byzantine adversaries, it is beneficial to introduce incentives for operators to join the system and act honestly. Moreover, operators can be compensated for on-chain transactions. Such incentives can be achieved by introducing execution fees paid by the users to the operators. We expect these fees to be significantly lower than Ethereum transaction fees since replication of computation is only required among a small pool. Additionally, registration fees for operators can be used to mitigate the risk for sybil attacks. By mitigating these

attacks and due to the random assignment of enclaves to contract pools, operators can only actively enforce centralization at high cost.

Efficiency Improvements. Instead of propagating each contract invocation, a more fine-grained distinction based on the action can be added. In particular, a simple state retrieval must not be propagated. In order to improve the efficiency of the manager, messages and responses are not stored persistently. Instead, only their hashes are stored and the actual data is propagated via events. Moreover, the total on-chain transactions can be reduced by letting the executor enclave challenge only the unresponsive watchdog enclaves.

VII. Security Analysis

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

A. Protocol Security

For the sake of brevity, we present the full security analysis of our *POSE* protocol including formal theorems in Appendix A. Here, we provide an intuition of our security guarantees.

The *POSE* protocol satisfies *correctness*, ϵ-*liveness* and *state privacy*.

(1) Intuitively, *correctness* means that an adversary cannot influence the smart contract execution within an enclave such that the result is invalid according to the contract logic. Our creation protocol ensures that all enclaves of a pool store the correct contract code. The TEE security guarantees and the *POSE* code ensure that each enclave executes the stored code correctly. Finally, the synchronization mechanism guarantees that each enclave is up-to-date with the blockchain up to some slack, τ^{off}_{slack}. This ensures that on-chain transactions are considered by the smart contract execution, at least after time τ^{off}_{slack}.
(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.

B. Architectural Security

We further examine the architectural security of enclaves. The case of a user or TEE operator going offline by turning off their machine is covered in the protocol security (cf. Section VII-A); here we focus on parties that follow the protocol, trying to gain an unfair advantage in various ways.

The adversary might try to perform a memory corruption attack on the client used by users to interact with the executor (e.g., to send inputs). To mitigate this risk, the software should be implemented in a memory-safe language, like Python or Rust, and be open source so that it can be easily inspected.

A malicious TEE operator can also try mounting a memory-corruption or a side-channel attack on its TEE. As mentioned in A1.1, we assume that the TEE protects the confidentiality of the enclave and prevents leakage. However, in practice, cache-based side-channel attacks have been successfully demonstrated also on ARM processors [44]. While we want to stress that our ARM TrustZone-based implementation is a research prototype and the design is TEE-agnostic, the risk of these attacks can be mitigated by making the TEE opt-out of shared caches and flush private caches upon context switch, as proposed in [19]. Alternatively, a more advanced TEE design can be used [24], [19], [16]. Moreover, if the enclave code has an exploitable memory-corruption vulnerability, it is possible to mount a memory-corruption attack against it. One way to mitigate this risk, and hence, realize our assumption A1.2, is to use a memory-safe language for our smart contracts (in our case, Lua), or to deploy a run-time mitigation (like CFI [11]). Yet, in practice, an adversary might still be able to compromise an enclave. In this case, only the contracts of this enclave are affected. The consequences depend on the role of the enclave: for an executor enclave, the adversary gets full control over the contract; for a watchdog enclave, the adversary can only break state privacy.

Finally, an adversary might build a malicious smart contract with the goal of compromising secrets owned by other contracts or blocking an enclave by entering into an infinite loop. We mitigate against the first scenario by ensuring that only one smart contract is executing at any given time in an enclave, so that no foreign plain text secrets are present in memory at any point during contract execution. In case of multiple enclaves running on the same system, the TEE is isolating enclaves from each other such that no contract can tamper with another (cf. assumption A1.1). To handle infinite loops, we leverage a Lua sandbox [14], which interrupts the execution of the Lua code after a predetermined number of instructions has been issued and disables access to unsafe functions and modules.

VIII. Implementation

In order to evaluate *POSE*, we implemented a prototype for the manager and the enclaves, which uses TrustZone for the enclaves themselves and Lua as the smart contract programming language. We open source our prototype implementation to foster future research in this area[5]. We describe each of them in the following.

Manager. For the manager we use an Ethereum smart contract written in Solidity, which we will refer to as *manager* in the

[5]https://github.com/AppliedCryptoGroup/PoseCode

following. Even if this implementation is based on Ethereum, we note that our design can be realized on any blockchain supporting rich smart contracts. The manager keeps a list of all registered enclaves in the network as well as a list of all deployed contracts, including their public information, e.g., the address of the current executor. As mentioned in the protocol described in Section V-E, the manager provides functions to register an enclave, create a new *POSE* contract, deposit or withdraw money, and functions to challenge the current executor or any of the watchdogs. To synchronize all participants, every time a challenge related function was called it will throw an appropriate Solidity event.

Enclaves. The contract creator, executor, and watchdogs are enclaves running in a TEE. As our protocol is TEE-agnostic and all commercial TEEs exceed smart contracts' on-chain requirements on memory/computational-power capabilities significantly, we chose to use ARM TrustZone [15] for our prototype. TrustZone features a traditional programming model (OS, and user-space applications with standard library), and the Open Portable Trusted Execution Environment (OP-TEE) OS [42] already supports a large fraction of standard functionality, and hence, does not force us to reimplement this for the contract execution environment. TrustZone supports two execution modes: secure world and normal world. The system's memory can be freely distributed among these worlds. The secure world is an trusted OS which is completely independent from the normal OS, which in our case is Linux. Code running in the secure world is called a *Trusted App* (TA). A TA may only communicate with the normal world via shared memory regions, which are explicitly allocated as such. We implement the *POSE* enclaves as TAs. Computations in the secure world have native performance; yet, switching between worlds has a constant but negligible overhead (in our tests around 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[6]. 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.

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

IX. Evaluation

This section examines *POSE* regarding complexity and performance. In the following, we will report absolute performance numbers and discuss these in relation to Ethereum itself, but also compare to existing works based on TEEs, namely FastKitten and Bitcontracts. FastKitten has a highly similar set of tested smart contracts, so a comparison can put our numbers in perspective. For Bitcontracts, we reimplemented Quicksort with the same experiment setup. Note, that the smart contracts can still be implemented differently, and the performance and the TEE differ.

Complexity. Running a *POSE* contract in the benign case, i.e., if all involved enclaves respond, requires exactly two blockchain interactions for the setup. Each user of a contract also needs one blockchain interaction each time the user deposits or withdraws money regarding the contract. However, as *POSE* does not require a fixed collateral for the setup, the money transactions do not inherently prevent the contract from execution—except the specific contract demands it. Otherwise, when either the executor or any watchdog fails to respond, each challenge requires two blockchain interactions. The delay incurred by our challenge protocol is dominated by the on-chain transactions. This holds also for other off-chain solutions, e.g., state-channels [46], [26], [22], Plasma [52], [37], Rollups [48], [5] and FastKitten [25]. For instance, the time it takes for an honest executor to kick a watchdog is $325s$ on average. We discuss timeout parameters and the challenge delay more thoroughly in Appendix D. In the worst-case, a malicious operator does not respond to the off-chain messages but to the challenges in every execution step, which would effectively reduce *POSE*'s execution speed beneath that of the blockchain. However, such an attack requires continuous blockchain interactions from the malicious party and hence entails costs for every execution step (cf. Section IX "Manager").

Test Setup. We deployed a test setup with our prototype implementation for performance measurements. The test setup consists of five devices. For the enclaves we deployed three Raspberry Pi 3B+ with four cores running at 1.4GHz. These are widely available and cheap devices that support ARM TrustZone. As state updates are small (just the delta to the previous state) and watchdogs receive and process the state updates in parallel, we do not expect an increase of the pool size to significantly influence the evaluation. Further, we used `ganache-cli` (6.10.2) to emulate a Ethereum blockchain in our local network, which runs the Solidity contract that

TABLE I. COST OF EXECUTING THE *POSE* MANAGER. THE USD COSTS WERE ESTIMATED BASED ON THE PRICES (GAS TO GWEI AND ETH TO USD) ON MAY. 8, 2022 [27], [21]. *FOR COMPARISON, THESE ARE THE COSTS OF POPULAR OPERATIONS ON ETHEREUM.

Method	Cost	
	Gas	USD
`registerEnclave`	175 910	13.23
`initCreation`	198 436	14.91
`finalizeCreation`	79 545	5.98
`deposit`	37 255	2.80
`withdraw`	36 997	2.78
`challengeExecutor`	54 654	4.11
`executorResponse`	51 478	3.87
`executorTimeout`	53 327	4.01
`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

implements the manager. Finally, a fifth device emulates multiple users by simply sending out network requests to both the manager and enclave operators, which are all connected via Ethernet LAN.

Manager. As the *POSE* manager is implemented as an Ethereum smart contract, interactions with it incur some costs in the form of Gas. The costs of all implemented methods of the Solidity contract are listed in Table I. The first five methods are used for benign *POSE* contract execution. The second part of the table shows methods that are required for challenges, including the response and timeout methods to resolve them. In terms of storage, each additionally registered enclave will require 64 bytes and each contract 288 bytes + (pool size $\times$ 32 bytes) of on-chain storage.

Contract Execution. To measure and demonstrate the efficiency of *POSE* contract execution, we implemented three applications as Lua code in our test setup. All time measurements are averaged over 100 runs. Regardless of the used contract, setting up an executor or watchdog enclave with a Lua contract takes 189ms. Creating an attestation report for the enclave takes another 367ms with OP-TEE's built-in remote attestation using a one-line dummy contract. For our biggest contract, Poker, the attestation takes 377ms, resulting in a total setup time of 566ms. In contrast, FastKitten needs 2s for enclave setup. Note that FastKitten needs an additional blockchain interaction. Multiple contracts run by a single operator are executed in parallel, including network communication. Thus, the number of enclaves, contracts and transactions a single operator can process depends on the operator's hardware. As modern servers CPUs feature 128 cores [23], and servers often feature multiple CPUs, we do not expect parallel execution to affect performance significantly. However, to prevent overload, the number of pools an operator participates in can be limited.

Rock paper scissors. This is an implementation of the popular game with two players. Unlike traditional smart contracts, we can leverage *POSE*'s private state to simply store each player's input, instead of having to use much more complex multi-round commitments. The resulting smart contract is 27 lines of code (LoC). Disregarding the delay caused by human players, the execution time of one round with two user inputs is 32ms. In comparison, FastKitten only needs 12ms, but is also running on a much more powerful machine. In contrast, executing this game on Ethereum would take around 5 minutes for each round (20 confirmation blocks, 15s block time each).

Poker. We have also implemented Poker as a multi-party contract running over multiple rounds. Note that in *POSE*, the poker game can be implemented as an ongoing cash game table, i.e., players may join or leave the table at any time, as contracts in *POSE* do not have to be finite. Each round consists of three phases each requiring an input from all users. The resulting smart contract is 209 lines of code (LoC). We execute the contract with five players who have their deposit ready at the start, with a total execution time of 199ms (vs. 45ms in FastKitten, but again, on a more powerful machine). Playing this game on Ethereum would take 5 minutes per player input.

Federated Machine Learning. For this application, users can submit locally trained models, which will be aggregated to a single model by the contract. Any user can then request the new model from the contract. For our measurements, each user trained a convolutional neural network consisting of 431 080 individual weights on the MNIST handwritten digits dataset [62]. For aggregation, the contract averages every existing weight with the corresponding weight sent by the user. The smart contract itself is only 5 LoCs, as we load the existing weights separately. Each aggregation took 238ms, which demonstrates the efficiency of *POSE*. Trying to execute the same function on Ethereum, for each aggregation, storage of the weights alone would exceed 1 billion gas (assuming 4 bytes float per weight) and the calculation over 3.4 million gas (8 gas per weight).

Quicksort. We have also implemented Quicksort to sort a hardcoded input array of 2048 random integers, as done in Bitcontracts [59]. The resulting smart contract is 29 lines of code (LoC). The total execution time of the contract is 20ms. Compared to the 6ms in Bitcontracts, we use a less powerful machine (Bitcontracts uses an AWS T2.micro instance with a recent Intel processor at 3.3Ghz), while our performance measurement also includes additional steps like context switches and the setup of the enclave runtime. Executing this Quicksort contract on Ethereum would cost around 6.5 million gas.

Watchdog State Updates. When an executor operator has been dropped, a watchdog takes over execution. For this to work, state changes are distributed to the watchdogs. Storing the current state and restoring it on a watchdog takes 17ms for the poker contract (averaged over 100 runs, corrected for network latency), which also has the biggest state among the ones we implemented.

Enclave Teardown. After an executor enclave is not expecting further inputs and finished the smart contract execution, the execution environment has to be cleaned up for the next smart contract, i.e., cryptographic secrets and the smart contract in the shared memory need to be zeroed. This takes 25ms.

X. RELATED WORK

Ethereum [58] is the most prominent decentralized cryptocurrency with support for smart contract execution. However, it is suffering from very high transaction costs and data used by smart contracts is inherently public.

TABLE II. OVERVIEW OF RELATED WORK, n IS #TRANSACTIONS.

	No collateral	Private state	Blockchain interactions (optimistically)	Non-fixed lifetime & group
Ethereum [58]	✓	✗	$O(n)$	✓
MPC [40], [41], [39]	✗	✓	$O(1)$	✗
State Channels [46], [26], [22]	✗	✗	$O(1)$	✗
VM-based [36], [60], [59]	✗	✗	$O(n)$	✓
Ekiden [20]	✗	✓	$O(n)$	✗
FastKitten [25]	✗	✓	$O(1)$	✗
POSE	✓	✓	$O(1)$	✓

Hawk [38] aims for improving the privacy by automatically creating a cryptographic protocol from a high-level program in order to allow computation on private data without disclosing it. However, this complex cryptographic layer further decreases performance of the system and increases costs. Similarly, approaches based on Multiparty Computation (MPC) [40], [41], [39] distribute the computation between multiple parties such that no party can access the cleartext data. These approaches have substantial overhead in performance, communication and collateral required.

One approach to alleviate the complexity limitation are state channels [46], [26], [22], which enable parties to lock some funds on the blockchain, execute complex contracts off-chain, and finally commit the results of the contract to the blockchain. This is efficient if all parties agree on the results; otherwise, the dispute can be solved on-chain, which takes longer and is more expensive.

Arbitrum [36] represents a smart contract as a virtual machine (VM), which is executed privately by a number of "managers". After execution, if all managers agree on the result of the computation, this result can be simply signed and committed to the blockchain, without the need to perform the computation on chain. In case managers disagree, a bisection algorithm is used to compare subsets of the execution on chain and find which is the first instruction on which the managers disagree, then punish the malicious manager(s). Hence, as long as at least one manager is honest, the correct result is computed. While computationally efficient, this on-chain protocol is still relatively expensive, so Arbitrum also includes financial incentives to encourage the managers to behave. The managers have full access to the VM's data, so confidentiality is broken if even one manager is malicious. Unlike Arbitrum, *POSE* does not require multiple parties to execute the smart contract: the watchdog enclaves just need to acknowledge the new states, unless the executor enclave fails.

ACE [60] and Bitcontracts [59] are similar to Arbitrum, but they allow the results of contract executions to be approved by a configurable quorum of service providers, not necessarily all of them. Unlike *POSE*, ACE does not support private state and requires on-chain communication per contract invocation. Although the transaction is computed off-chain, the invocation and the result are registered on-chain. Further, Arbitrum and ACE require changes to the blockchain infrastructure, hence, they are harder to deploy in practice.

Ekiden [20] is also an off-chain execution system that leverages TEE-enabled *compute nodes* to perform computation and regular *consensus nodes* that interact with a blockchain. The major drawback of Ekiden is that it requires every computation step to retrieve its initial status from the blockchain, and it only supports input from one client at a time. Moreover, the atomic delivery of the output of each step requires to wait for publication of the updated state before the output is made available to the client. Hence, any highly interactive protocol with multiple participants (e.g., a card game) would incur significant delays between turns just to wait for the blockchain. The paper evaluates on a fast blockchain, Tendermint, but does not quantify its latency for interactive protocols on mainstream blockchains like Ethereum. The Oasis Network uses an updated version of Ekiden [30]; yet, this version still requires to store state on the blockchain after each call.

FastKitten [25] also leverages TEEs to perform off-chain computation. It assumes a rational attacker model, with financial incentives to convince all participants to follow the protocols. If they all do, the communication happens directly between the TEE and them, thus dispensing with the high latency due to blockchain roundtrips. However, FastKitten only supports contracts with a predefined list of participants and a limited lifespan. It also requires the TEE operator to deposit as much as every participant combined as collateral. *POSE* lifts those restrictions: it enables long-lived smart contracts with an unknown set of participants and requires no collateral from the TEE owners. Further, *POSE* achieves strong liveness guarantees in the presence of byzantine adversaries, while FastKitten assumes a rational adversary.

ROTE [45] is an approach to detect rollback attacks on TEEs by storing a counter on other TEEs. This approach is similar to the watchdog enclaves used in *POSE* to ensure that execution of a smart contract continues. However, unlike *POSE*, ROTE can only detect rollback attacks, but cannot prevent malicious operators from withholding the state. SlimChain [61] primarily aims at reducing on-chain storage, while still requiring blockchain interactions to store state commitments. Further, the paper does not address storage nodes crashing, which would lead to a liveness violation. Pointproofs [33] proposes a new vector commitment scheme to reduce the storage requirements on blockchain validators. Although validators do not need to store all values of a smart contract, once a transaction provides these values, the execution is still performed on-chain. In contrast, *POSE* works entirely off-chain in the optimistic case and ensures liveness.

Chainspace [12] proposes an entirely new distributed ledger platform focusing on sharding combined with a directed acyclic graph structure, while POSE extends established blockchains (e.g., Ethereum). ResilientDB [54] proposes a consensus protocol that clusters validators' geo-location to minimize network overheads. In contrast, *POSE* is 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).

XI. Conclusion

Smart contracts have become an indispensable tool in the era of blockchains; yet, current approaches suffer from various shortcomings. In this paper, we introduce *POSE*, a novel off-chain execution protocol that addresses all of these shortcomings to enable much more versatile smart contracts. We showed *POSE*'s security and demonstrated its feasibility with a prototype implementation.

Acknowledgements

This work was supported by the European Space Operations Centre with the Networking/Partnering Initiative, the German Federal Ministry of Education and Research within *Sanctuary* (16KIS1417) and within the *iBlockchain project* (16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 – 236615297 (CROSSING Project S7)*, by the European Union's Horizon 2020 Research and Innovation program under Grant Agreement No. 952697 (ASSURED), by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*.

References

[1] Cardano. https://cardano.org/. (Accessed on 05/20/2021).

[2] Cryptokitties - collect and bread furrever friends! https://www.cryptokitties.co/. Accessed 14-08-2022.

[3] Enhanced attestation (v3). https://docs.samsungknox.com/dev/knox-attestation/about-attestation.htm. Accessed 20-04-2022.

[4] Etherscan - ethereum average block time chart. https://etherscan.io/chart/blocktime. Accessed 20-09-2021.

[5] Optimistic rollups - ethhub. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/optimistic_rollups/. (Accessed on 05/20/2021).

[6] Qualcomm® trusted execution environment (tee) v5.8 on qualcomm® snapdragon™ 865 security target lite. https://www.tuv-nederland.nl/assets/files/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.

APPENDIX

A. Protocol Security

We analyze the security of our protocol under the assumption of an IND-CPA secure encryption scheme, an EU-CMA secure signature scheme and a collision resistant hash function in the following. We present definitions of correctness, ϵ-liveness and state privacy.

1) Correctness: We define a state update as the evaluation of a transition function f, which receives as inputs a user U, a user input $move$ and a copy of the blockchain BC. The *correctness* property states that each state update evaluates the transition function as defined by the contract code with valid inputs, i.e., U is the (potentially malicious) client triggering the transition, $move$ the input of U and BC a valid copy of the blockchain that is at most τ_{slack}^{off} behind the main chain.

Claim 1 (Correctness)**:** *POSE satisfies correctness.*

We first note that according to our adversary model, a corrupted operator may delete any message intended for her enclave or generated from her enclave. However, the correct execution of the *POSE* program inside the enclave cannot be influenced. When an operator creates a *POSE* enclave, the registration process ensures that the new enclave indeed runs the *POSE* program. To this end, our protocol utilizes the TEE attestation mechanism, which generates a verifiable statement that the enclave is running a specific program. Upon registration with the manager M, M checks the validity of the attestation statement as well as the blockchain evidence, the signed hash and number of the latest block known to the enclave. M only registers the enclave in the system if the new enclave is running the *POSE* program and is not further behind than maximally τ_{slack}^{off}. Finally, the TEE integrity and confidentiality guarantees ensure that a malicious operator cannot modify the enclave's code, tamper with its state or access its private data, in particular, its signature keys.

During the creation of a contract, the pool enclaves attest the code of the installed contract to the creation enclave. The creator checks that the code is consistent with the hash stored in the manager before signing a creation confirmation. Hence, it is not possible, without breaking the EU-CMA security of the signature scheme or the collision resistance of the hash function, to create a valid creation confirmation for a contract with different code than specified by the creation request.

Next, contract state updates can only be triggered by invoking the executor enclave with an execution request or invoking a watchdog enclave with an update request. The correctness of the latter is reduced to the correctness of the former. To see this, we observe that any update request to a watchdog enclave requires to be signed by the executor enclave. Clearly, the executor enclave only signs updates corresponding to its own executions. Therefore, an adversary cannot forge incorrect update request without breaking the unforgeability of the signature scheme. Also, the executor enclave can only issue a new state update if all watchdogs confirmed the previous one. Hence, it is not possible to tamper with the order in which the update requests are provided to a watchdog enclave. As stated before, the TEE integrity guarantees ensure the correct execution of the program code and hence the correct execution of the smart contract. It follows that a state update can only be achieved by providing inputs to the executor enclave. The executor enclave receives a signed message containing the action $move$ from user U and the relevant blockchain data from its operator. In Section V-D, we describe how our protocol achieves secure synchronization between the executor enclave and the blockchain. In particular, the synchronization mechanism ensures that the blockchain data accepted by an enclave is correct and complete in regard to a correct blockchain copy that is at most τ_{slack}^{off} behind the main chain. This guarantees that BC, represented by the received blockchain data, is a synchronized copy of the current blockchain. In order to protect inputs by honest users U, $move$ needs to be signed by U. This means an adversary cannot tamper with the input without breaking the signature scheme.

Finally, we note that each *POSE* enclave maintains a list of received messages. Since an honest user randomly selects a fresh nonce for each execution request, replay attacks can be detected and prevented by any executor enclave.

2) Liveness: The liveness property states that every contract execution initiated by an honest user U will eventually be processed with high probability. For a successful execution, a valid execution response is given by the executor. Unsuccessful execution can only happen in case of a contract *crash*. In this event, the contract execution halts and neither honest nor malicious users can perform successful contract executions anymore. We emphasize that the pool size can be set such that crashes happen only with negligible probability. In particular, for ϵ-liveness, the probability of a crash is bounded by $1 - \epsilon$.

Claim 2 (ϵ-Liveness)**:** *Let n be the total number of enclaves in the system, m be the number of malicious operators' enclaves and s be the contract pool size. POSE satisfies ϵ-liveness for $\epsilon = 1 - \Pi_{i=0}^{s-1}(\frac{m-i}{n-i}) > 1 - (\frac{m}{n})^s$.*

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 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.

3) State Privacy: The *state privacy* property says that the adversary cannot obtain additional information about a contract state besides what she learns from the results of contract executions alone.

Claim 3 (State Privacy)**:** *POSE satisfies state privacy.*

The smart contract's state is maintained by the enclaves within the execution pool. According to our adversary model (see Section II), the TEE provides confidentiality guarantees, i.e., the execution of an enclave does not leak any data. Hence, the smart contract's state is hidden from the adversary, even if the enclave's operator is corrupted. The only point in time when information about the contract's state is revealed is at the end of the execution protocol. However, the data provided as a result contains only public state and hence does not reveal anything about the private state. During the execution protocol, the executor enclave propagates the new state to all watchdog enclaves. However, the transferred data is encrypted using an IND-CPA secure encryption scheme. The security of the scheme guarantees that an adversary seeing the message cannot extract information from it. While an enclave only publishes outputs after successful executions, we need to show that each produced output is final. In particular, a succeeding executor must not be able to revert to a state in which a published output should not have been produced. To this end, the state of the executor enclave producing a particular output needs to be replicated among all other enclaves before revealing the actual output. This property is achieved by the state propagation

mechanism of *POSE*. An enclave only returns an output if all enclaves in the pool confirm the corresponding state update. The EU-CMA secure signature scheme guarantees unforgeability of the confirmations. Hence, each confirmation guarantees that the corresponding enclave has updated its state correctly. Further, the correctness property of our protocol (cf. Section A1) ensures an enclave is always executed with a correct blockchain copy; thus, is always aware of the correct pool composition. This means an output can only be returned if the whole pool received the corresponding state update.

B. Supported Contracts

POSE supports contracts with a dynamic set of users of arbitrary size and an unrestricted lifetime. The timeouts need to be set reasonable with respect to the expected execution time of the contracts to allow the execution of complex contracts and to prevent denial of service attacks at the same time. Interaction between *POSE* contracts can be realized by letting the TEE of the calling contract instruct its operator to request an execution of the second contract via the respective executive operator and wait for the response. We deem the exact specification, e.g., enforce an upper bound on (potentially recursive) external calls to guarantee timely request termination, an engineering effort. Calls from POSE contracts to on-chain contracts can be supported similarly to our payout concept (Appendix E).

C. Further Protocol Blocks

To keep the specification of the *POSE* protocol in the main body simple and compact, we have excluded the formal specification of the creation process and the validation algorithms. In this section, we present the validation algorithms. For the formal specification of the creation process, we refer the reader to the full version of the paper [31].

All of the different messages sent to the manager throughout the protocol need to be validated with several checks. In order to keep the description compact, we did not include the validation steps in the protocol figures but extracted them into a validation algorithm specified in Program 3. The algorithm is invoked with an counter specifying the checks that should be performed, an optional message that should be checked and the contract state tuple maintained by the manager. The validation returns `ok` if all requirements are satisfied and M can continue executing and `bad` if M should discard the received request.

D. Timeouts

Our protocol incorporates several timeouts δ^*_{off}, which define until when an honest user or operator expects a response to a request, and δ^*_{on}, which define until when the manager expects a response to a challenge. These timeouts have to be selected carefully s.t. each honest party has the chance to answer each message and challenge before the respective timeout expires. In this section, we elaborate on the requirements on the timeouts. We neglect message transmission delays and also assume that each challenge sent to the manager will directly be received by all operators (already before it is included into a final block)[7]. We recall the maximum blockchain delay which is defined as $\delta_{\mathsf{BC}} = \alpha \cdot \tau$ (cf. II and IV). 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. IV). 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 V-F. 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 timeout allows the executor to provide the kick transaction together with enough confirmation blocks to

[7] 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.

Program 3: Algorithm *Validate*

The validation algorithm performs the following checks. If input $C = \bot$, 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 $C.creator = \bot$, $C.c1Time = \bot$ and $Verify(m) = \texttt{ok}$.
- On input $(2, res; C)$, parse res to $(\texttt{ok}, id, \cdot, h; T)$. Require that $C.creator = \bot$, $H(C.c1Msg) = h$, $C.c1Time + \delta^1_{on} > \mathsf{BC}.now$, $Verify(res) = \texttt{ok}$ and $C.pool[0] = T$.
- On input $(3; C)$, require that $C \neq \bot$, $C.creator = \bot$, $Cc1Msg \neq \bot$ 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 $C.creator = \bot$, $C.c2Time = \bot$, $C.pool[0] = T$ 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 $C.creator = \bot$, $C.c2Time + \delta^2_{on} > \mathsf{BC}.now$, $Verify(conf) = \texttt{ok}$, $h = h'$ and $T \in C.pool$.
- On input $(6; C)$, require that $C.creator = \bot$, $C.c2Time \neq \bot$ and $C.c2Time + \delta^2_{on} \leq \mathsf{BC}.now$.

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[8]. 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

[8]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).

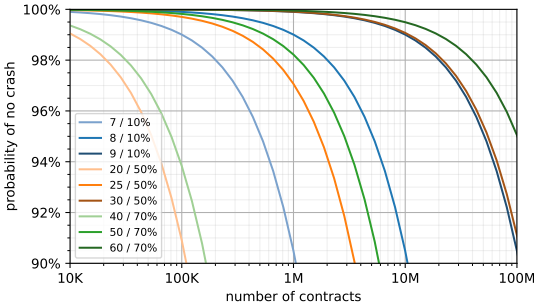


Fig. 5. Cumulative probabilities of no contracts crashing w. large number of POSE contracts for different pool sizes s and adversary shares m, "s/m".

POSE contract, identified by id, a user U sends a message $(\texttt{deposit}, id, amount; U)$ 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 $Verify(\texttt{deposit}, id, amount; U) = \texttt{ok}$. 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. V-D) and the enclave forwards 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.

D

OFFLINE MODEL GUARD: Secure and Private ML on Mobile Devices (DATE'20)

---

[17] Sebastian P. Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. Offline Model Guard: Secure and Private ML on Mobile Devices. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 460–465. IEEE, 2020. https://ieeexplore.ieee.org/document/9116560. CORE Rank A*. Section 4.2.

This paper can be found in the Versions of Record in the ACM Digital Library:

`https://ieeexplore.ieee.org/document/9116560`

DOI: 10.23919/DATE48585.2020.9116560

OFFLINE MODEL GUARD: Secure and Private ML on Mobile Devices

Sebastian P. Bayerl*, Tommaso Frassetto†, Patrick Jauernig†, Korbinian Riedhammer*, Ahmad-Reza Sadeghi†, Thomas Schneider†, Emmanuel Stapf†, Christian Weinert†
*Technische Hochschule Nürnberg, Germany, {sebastian.bayerl, korbinian.riedhammer}@th-nuernberg.de
†Technische Universität Darmstadt, Germany, {tommaso.frassetto, patrick.jauernig, ahmad.sadeghi, emmanuel.stapf}@trust.tu-darmstadt.de, {schneider, weinert}@encrypto.cs.tu-darmstadt.de

***Abstract*—Performing machine learning tasks in mobile applications yields a challenging conflict of interest: highly sensitive client information (e.g., speech data) should remain private while also the intellectual property of service providers (e.g., model parameters) must be protected. Cryptographic techniques offer secure solutions for this, but have an unacceptable overhead and moreover require frequent network interaction.**

In this work, we design a practically efficient hardware-based solution. Specifically, we build OFFLINE MODEL GUARD (OMG) to enable privacy-preserving machine learning on the predominant mobile computing platform ARM—even in offline scenarios. By leveraging a trusted execution environment for strict hardware-enforced isolation from other system components, OMG guarantees privacy of client data, secrecy of provided models, and integrity of processing algorithms. Our prototype implementation on an ARM HiKey 960 development board performs privacy-preserving keyword recognition using TensorFlow Lite for Microcontrollers in real time.

***Index Terms*—TEE, TrustZone, private ML, speech processing**

I. INTRODUCTION

An increasing number of applications running on mobile devices like smartphones and tablets relies on machine learning (ML) services to enhance the user experience, e.g., to give an estimate on battery life based on user behavior, improve image quality, or perform speech recognition.

Many of these ML services require frequent cloud interaction, resulting in severe privacy risks for billions of users due to the highly sensitive nature of such remotely processed data. Besides potentially confidential and intimate content, voice recordings, for example, contain unique biometric information that can be abused, e.g., for impersonation attacks and distributing fake recordings.

Privacy breaches in this domain are not fiction: in 2018, a customer requested his recording archive from Amazon, but accidentally got access to 1,700 audio files from a stranger [1]. Furthermore, state authorities ordered Amazon to hand out recordings as they might contain evidence of crime [2]. Media reports also revealed that Apple, among others, sent voice recordings to third party companies in order to improve their service with manual transcriptions. The employees of those companies got to listen to private discussions between doctors and patients, business deals, criminal dealings, and sexual encounters [3]. Moreover, biometric data used for identification was recently leaked at a large scale: the database of a UK government contractor with more than a million fingerprints and facial recognition information was publicly accessible [4].

When relying on online services for mobile ML applications, there are also usability issues to consider: high latency and, therefore, a bad user experience occurs if the user has an unreliable or low-bandwidth network connection, and high roaming fees may apply if the user is abroad.

A trivial solution for all these issues is to process all sensitive user data on the client's device. Previously, this approach was severely limited by the storage space constraints on mobile devices and the storage space requirements of ML models used in practice. Recently, though, Google lifted this limitation by training a recurrent neural network (RNN) model for character-level speech recognition and compressing it to only 80 MB, while delivering the same accuracy as former cloud-based production models with a size of multiple gigabytes [5], [6].

However, deploying such a model in unencrypted form is often not in the interest of the service provider. A production-level model constitutes intellectual property as the underlying training data is usually hard to obtain and creating an accurate while compact model requires extensive expertise [7]. Furthermore, if attackers have unrestricted model access, the privacy of people represented in the training data is even more threatened by, e.g., membership inference attacks [8] and unintended memorization [9].

Cryptographic techniques like homomorphic encryption (HE) and secure multi-party computation (SMPC) provide solutions for this conflict of interest: with HE, private inputs can be securely processed under encryption by the client or the service provider, whereas with SMPC, client and server can jointly compute any function on private inputs in a provably secure protocol. Unfortunately, the computational overhead for HE when performing complex ML tasks is impractical for the given mobile scenario, whereas the amount and the frequency of required network communication is the bottleneck for SMPC protocols. Thus, we explore hardware-assisted solutions to deliver secure and private ML on mobile devices in offline scenarios while providing practical efficiency.

Our Contributions. In this work, we build OFFLINE MODEL GUARD (OMG), a generic architecture that efficiently protects machine learning tasks on mobile devices like smartphones and tablets, and demonstrate its practicality using offline keyword recognition as an example application.

978-3-9819263-4-7/DATE20/©2020 EDAA

OMG leverages unprivileged (normal-world) user-space enclaves on ARM platforms to execute ML tasks in a hardware-protected environment that is two-way isolated from all other system components to minimize the attack surface. Utilizing TrustZone functionality, OMG can securely access peripherals like the microphone to protect sensitive information directly from the source. As a result, OMG guarantees complete privacy of client data, secrecy of the provided ML models, and integrity of processing algorithms.

We provide a fully functional prototype implementation of OMG on an ARM HiKey 960 development board for offline keyword recognition based on TensorFlow Lite for Microcontrollers [10]. As TrustZone on ARM does not provide user-space enclaves, we leverage SANCTUARY [11] for our implementation. Our performance evaluation demonstrates that secure and private offline speech processing is possible in real time even with strong protection guarantees. As we developed our prototype with TensorFlow compatibility in mind, our implementation can easily be extended to network architectures used for other related tasks such as end-to-end continuous speech recognition, speaker verification, and emotion recognition.

II. Related Work

In the following, we review existing works that preserve privacy in machine learning. The goal there is usually to train a model on the server side without allowing the server to see training data in the clear, or to obliviously classify input data without leaking the model (inference). Proposed solutions either rely entirely on cryptography or build on TEEs.

For protecting only the IP of ML models there also exist orthogonal works for model watermarking [12] and fingerprinting [13] that do not consider the privacy of client inputs.

A. Cryptography

The cryptographic techniques used for privacy-preserving machine learning are homomorphic encryption (HE) and secure multi-party computation (SMPC). Also, combinations of these techniques are being studied. HE allows to perform operations directly on encrypted data, but generally incurs a high computational overhead. SMPC allows multiple parties to jointly perform secure computations on shared data. This works by obliviously evaluating a Boolean or arithmetic circuit representation of the desired functionality, but results in a high communication overhead and for some protocols requires interaction for each layer of the circuit.

For cryptographic protocols it is possible to formally prove security with respect to input privacy. However, many protocols and corresponding implementations assume that both client and server honestly follow the protocol description. This assumption is unrealistic in real-world scenarios since mobile clients might run modified applications. Securing such protocols against malicious parties comes at additional cost.

Privacy-preserving neural network inference via HE and SMPC was studied in [14]–[16]. Thereafter, many frameworks for privacy-preserving machine learning have been developed, e.g., [17]–[22]. They allow at least for secure deep/convolutional neural network inference and are usually benchmarked with standard image classification tasks.

Using such cryptographic frameworks requires expert knowledge and thus they are hardly accessible for ML experts. However, recently there are efforts to integrate cryptographic protocols into standard ML tools: for TensorFlow there are HE [23] and SMPC [24] implementations, and for Intel's ngraph compiler there exists HE support [25].

Unfortunately, the current performance results discourage from actual deployment and scaling them to more involved speech processing tasks seems unrealistic [26]. Addressing all outlined disadvantages, with OMG we propose a computation- and communication-efficient hardware-assisted design for secure and private ML on mobile devices that enforces correct execution of the algorithms and can easily be used by ML experts due to TensorFlow Lite compatibility.

B. Trusted Execution Environments (TEEs)

Compared to cryptographic techniques, trusted execution environment (TEE) architectures provide several orders of magnitude better performance for protecting ML services [27]. Most of the existing works rely on Intel SGX as the dedicated TEE architecture to protect ML services.

Ohrimenko et al. [28] protect ML algorithms and models in SGX enclaves. They consider a scenario where sensitive data from multiple data providers is aggregated on a remote server while SGX enclaves are used to protect the training process. However, the enclaves might leak information to the untrusted software on the server through data-dependent access patterns, which can be exploited in controlled-channel attacks [29], [30]. Therefore, the authors develop data-oblivious variants of standard ML techniques, e.g., support vector machines, neural networks, and decision trees, which guarantee that all memory accesses do not depend on secret data.

In Chiron [31], an ML-as-a-Service (MLaaS) scenario is considered where sensitive data is collected from customers and used for training without revealing the data to the MLaaS provider. This is achieved by performing the training process in a Ryoan [32] sandbox (based on SGX), which protects sensitive customer data but still offers the service provider the possibility to freely select, configure, and train the models.

Myelin [33] provides security guarantees similar to [28] as it relies on data-oblivious deep learning algorithms: every model owner compiles its deep learning model into a privacy-preserving model graph, which is then trained on a remote server (inside an SGX enclave) on sensitive data.

In [34], the authors introduce an alternative protection mechanism against controlled-channel attacks that is more efficient and suitable for real-time data processing. The authors propose to add noise to memory traces by accessing dummy data instead of enforcing data-oblivious memory accesses.

VoiceGuard [35] targets the use case of privacy-preserving speech processing. For this, sensitive voice recordings are collected from user devices, e.g., smart home devices like Amazon Echo, Google Home, and Apple HomePod, and are sent

via secure channels to a service provider. The service provider performs speech recognition using proprietary models provided by ML specialists in an SGX enclave, thereby protecting the user data as well the proprietary models. The inference results are then securely sent back to the user device. Very recent work [36] also enables efficient private online speech recognition but uses obfuscation techniques and the notion of differential privacy, which significantly degrades accuracy.

In contrast, MLCapsule [37] considers an offline MLaaS scenario where the trained model is used on the client side for inference while being protected using an SGX enclave.

None of the previous works considers the challenge of how user data can be securely collected on the user device. Intel SGX, which is mostly used as the dedicated TEE architecture, is not able to provide a secure communication channel from enclaves to system peripherals, e.g., the microphone or camera [38]. Thus, sensitive user data is endangered as it could be exfiltrated by malicious software running on the client device. With OMG, we present the first TEE architecture that provides protection for proprietary ML models and privacy-sensitive user input at the same time. Furthermore, while Intel SGX is a TEE widely available in recent Intel CPUs, most mobile devices like smartphones and tablets come with CPUs based on the ARM platform. This prevents using the previously proposed SGX-based solutions for securing relevant use cases on mobile devices, e.g., offline speech recognition. Thus, in this work, we present OMG for ARM-based devices and as an example application demonstrate privacy-preserving offline keyword recognition in real time.

III. Background

In the following, we introduce relevant details regarding the ARM TrustZone TEE implementation and the SANCTUARY security architecture [11] for user-space enclaves.

A. ARM TrustZone

Trusted execution environments (TEEs) combine memory isolation techniques [39]–[41] and attestation [42] with isolated execution to provide protected execution of security-critical code. For mobile devices, the predominant computing platform is ARM, which provides a TEE implementation called ARM TrustZone [43]. A chip with TrustZone capabilities simultaneously runs two security contexts (or "worlds") as virtual processors: a "normal world" and an isolated "secure world" (cf. Fig. 1). While the normal world executes a commodity OS (e.g., Android) and ordinary applications, the secure world forms a TEE for running security-critical code on a trusted OS.

A major assumption of TrustZone is that an attacker cannot compromise code running in the secure world. Unfortunately, the TrustZone design is flawed in this aspect: the isolation between applications in the secure world is rather weak and the attack surface is massively increased the more applications run therein [44]. Thus, the secure world with its privileged platform access is an attractive target for adversaries.

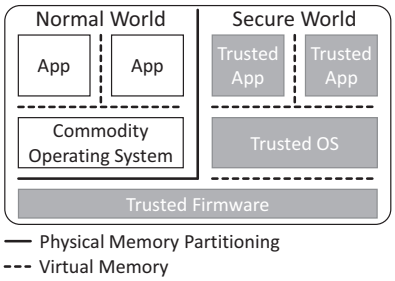


Fig. 1: ARM TrustZone architecture overview.

B. SANCTUARY

SANCTUARY [11] is a security architecture that circumvents the previously explained flaws of ARM TrustZone without requiring hardware extensions, heavy modifications of existing code bases, or major changes in the commodity OS. In particular, it allows to run security-critical code in user-space enclaves or so-called SANCTUARY Apps (SAs). SAs are executed in a normal-world environment that is protected via strict hardware-enforced two-way isolation from all other system components to minimize the attack surface. This is achieved by leveraging TrustZone's address space controller (TZASC) to exclusively bind memory to a (temporarily) dedicated CPU core running an SA.

The life cycle when running an SA is as follows:

1) Setup: Memory for the SA instance is prepared by loading the SANCTUARY library (SL), which is implemented using the Zircon microkernel [45], and the SA. The TZASC is securely configured to isolate this memory region and the least busy CPU core is shut down. Besides the isolated memory, additional memory regions are shared with the commodity OS and the secure world, which allows the SA to access the secure world and (untrusted) OS services.
2) Boot: The memory is attested and the CPU core is booted with the SL providing a basic execution environment.
3) Execution: The SA runs as a normal-world user process, potentially using services provided by the commodity OS or secure world code.
4) Teardown: The CPU core is shut down, data in the first level cache (L1) is invalidated, the SA memory is cleaned and unlocked, and finally the CPU core is handed back to the commodity OS.

SANCTUARY provides code and data integrity as well as data confidentiality, is secure against malicious SAs, and has no negative impact on the user experience due to the wide availability of multicore chips for mobile devices. Furthermore, side-channel attacks that extract secrets from caches can be prevented easily since the L1 cache is core exclusive and the shared second level cache (L2) can be excluded from SANCTUARY memory without severe performance impact [11].

SANCTUARY extends TrustZone to provide an arbitrary number of user-space enclaves. Additionally, SANCTUARY inherits many useful features from TrustZone like secure boot or DMA attack protection. Moreover, TrustZone allows to assign sensitive peripherals exclusively to the secure world.

An SA can use this feature by sending communication requests to the secure world code. After checking the permission rights of the SA, the secure world reads from the sensitive data and directly stores it in the memory region shared with the SA. Thus, performance overhead is only produced by the additional world switches between the SA and the secure world.

IV. Security Model and Assumptions

In this paper, we consider two parties collaborating to perform ML tasks on sensitive data provided by one party while protecting the intellectual property of the other party.

The *user U* provides input data to be processed. She is concerned about the privacy of the content to be processed (i.e., her inputs as well as outputs) and biometric characteristics potentially used throughout processing. Lastly, the user does not want to be traceable across multiple sessions.

The *vendor V* (who might act as the service provider) provides ML algorithms including corresponding models. The models constitute the vendor's intellectual property, hence the user must not be able to reverse engineer, share, or break the license check of these models.

Adversary Model. The adversary's goal is to extract sensitive information, i.e., the intellectual property of the vendor, the input and output of the user, or data that allows the adversary to identify or track the user. We assume that the adversary is in control of the user's device. The adversary has full control over the software running in the normal world of the user's device, including privileged software like the commodity OS. We assume that the adversary cannot perform hardware attacks, e.g., a physical side channel to extract secret keys. For the enclave we assume that all of SANCTUARY's defense mechanisms are in place, including hardware cache partitioning (for a detailed discussion see [11]).

V. OMG Design

OMG enables privacy-preserving and efficient offline execution of ML algorithms on untrusted ARM-based systems. For the sake of simplicity, we explain our solution based on the speech recognition scenario visualized in Fig. 2.

The vendor V's private input consists of a ML model. The user U's private input consists of voice recordings. In this example, the ML model is the vendor's intellectual property and any information about its architecture or trained weights must never be disclosed. The only output is the transcription, which is sent to the user.

OMG works in three phases: (I.) preparation, (II.) initialization, and (III.) operation. In the preparation phase, the enclave (containing the SL and SA) is loaded and attested to user U and vendor V. Then, V provides the encrypted ML model to the enclave. In the initialization phase, V sends the decryption key for the ML model so that the enclave can decrypt the model. Finally, in the operation phase, the enclave is ready to perform offline speech recognition. U sends her voice recordings to the enclave and receives respective textual output (which can be further processed into an action, as with virtual assistants). Next, we detail the individual phases:

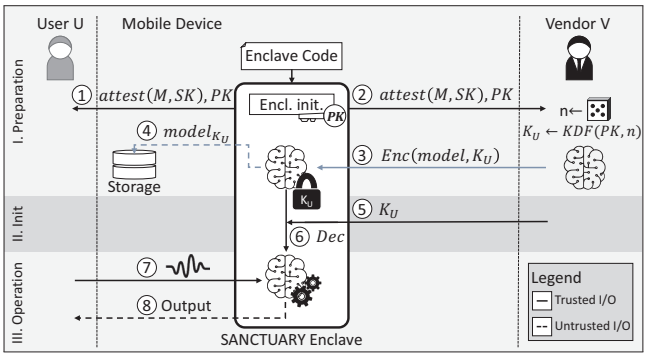


Fig. 2: OMG overview. Once the encrypted model is stored locally, steps in gray are optional until a model update.

I. Preparation Phase. First, the enclave needs to be run on U's device. The enclave contains the environment required to apply the ML model to input data. The enclave code can be open source, since it does not contain any vendor secrets (e.g., it may just consist of a TensorFlow environment), and can be distributed by the device manufacturer via regular distribution channels. To load the enclave, its code is first copied to memory and locked to a dedicated SANCTUARY CPU core so it cannot be changed anymore by the commodity OS (cf. § III-B). Then, the enclave is attested ("measured") by SANCTUARY, i.e., a cryptographic hash of the initial memory content of the enclave is created and stored securely. If the enclave code is manipulated before the creation process, the measurement will produce a different result and the manipulation will be detected.

SANCTUARY then assigns an unique asymmetric key pair to this enclave, e.g., by using RSA [46] (the public key *PK* is shown in Fig. 2). This key pair is derived from the platform certificate issued by the device vendor, effectively creating a certificate hierarchy similar to SSL certificates. To assure to U that the correct enclave code has been loaded, an attestation report is generated (i.e., the cryptographic hash of the initial memory content is signed using the secret key *SK* corresponding to *PK*) and sent to U using the secure output functionality of SANCTUARY ①. Such an attestation report is also sent to V using a secure connection (e.g., via TLS) directly from the enclave ②.

Note that the attestation report includes the enclave's public key *PK*. V uses *PK* and a nonce n to derive a symmetric encryption key K_U used only for this respective enclave and version of the model. V encrypts the ML model using K_U and securely provisions the model to the enclave ③.

The enclave then stores the model locally in unprotected storage ④. As the model can be loaded from untrusted local storage, after running the preparation phase once, steps ③ and ④ can be omitted until the vendor's model is updated.

II. Initialization Phase. Thanks to never making the decrypted model directly accessible to U, the initialization phase can be kept simple while providing strong guarantees to V. V can actively manage the access of U to the model by either sending or not sending the symmetric key K_U. In case

of, e.g., an expired license, V can stop sending K_U to the enclave, making it fail to decrypt the locally stored model. If V decides that U should be allowed to use the model, V securely sends K_U ⑤ to the enclave and the enclave decrypts the model ⑥. As the key K_U depends on the nonce n, this also prevents rollback attacks for U's locally stored model.

III. Operation Phase. In the operation phase, the actual ML task takes place. U can directly and securely provide voice recordings to the enclave as SANCTUARY allows secure input from peripherals like the microphone ⑦ by utilizing TrustZone features as described in § III-B. The speech data is then processed using the model, the output can be presented to the user or made available to other applications ⑧.

Once in the operation phase, the system can be queried repetitively, thereby avoiding repeated preparation and initialization costs as well as interaction with V. To do this, after a query is processed, the SANCTUARY core can be reallocated to the commodity OS while the memory is still locked such that no device or core is able to access it. When receiving a new query, a new SANCTUARY core is allocated and the locked memory is mapped to it for performing the ML task.

VI. Evaluation

We demonstrate the practicality of our approach by providing a fully functional prototype implementation of OMG on an ARM HiKey 960 development board based on TensorFlow Lite for Microcontrollers [10] and evaluating our prototype with an offline keyword recognition application.

The ARM HiKey 960 development board is equipped with an ARMv8 octa-core SoC (4 cores @ 2.4 GHz, 4 cores @ 1.8 GHz) with 3 GB of RAM, which closely resembles the specifications of today's mobile devices. We use such a development board instead of an off-the-shelf device since most vendors restrict developer access to TrustZone, which prevents us from setting up SANCTUARY (cf. § III-B). As our offline keyword recognition application is just a proof of concept, following [35], we do not focus on best accuracy, but study whether accuracy and runtime are affected when providing strong security guarantees.

The models are trained and evaluated on the Speech Command dataset [47] consisting of 105,000 WAVE audio files of people saying 30 different words. The recordings were post-processed to be a single word per file at a fixed 1 s duration.

We follow the TensorFlow Lite example recipe [10]: Features are computed using a 256 bin fixed point FFT across 30 ms windows (20 ms shift), averaging 6 neighboring bins, resulting in 43 values per frame. The 49 frames for each recording are concatenated, forming a fixed 49×43 compressed spectrogram ("fingerprint") per utterance.

The network architecture resembles [48], but is simplified to better match embedded requirements. The `tiny_conv` architecture feeds the audio fingerprint to a 2D convolutional layer (8 filters, 8×10, x and y stride of 2), followed by ReLU activation and a regular layer that maps to the output labels. During training, dropout is applied after the convolution layer.

TABLE I: Accuracy and runtime results for running the keyword recognition with and without OMG protection.

Model	Accuracy	Runtime
TensorFlow Lite "micro"	75 %	379 ms
TensorFlow Lite "micro" (OMG)	75 %	387 ms

We trained a system for a 12-class problem: *silence*, *unknown*, "yes", "no", "up", "down", "left", "right", "on", "off", "stop", "go". The model is first trained using TensorFlow and subsequently converted to a TensorFlow Lite and "micro" model. The resulting compressed model is about 49 kB in size.

We evaluated the "micro" model on a subset of the published test set comprising 10 examples for each class, excluding the two rejection classes "silence" and "unknown", since sensitivity for those would typically be tuned for production.

Inference was run on a 2.4 GHz core of the ARM development board both with and without OMG protection. Tab. I shows the overall accuracy for the 10 classes, and the respective runtimes in milliseconds. The accuracy with and without OMG protection is 75 %, confirming the correctness of the setup. The runtimes are very close when executed with and without OMG protection due to the fact that the hardware-enforced two-way isolation provided by SANCTUARY adds no additional overhead during execution. Since the overall duration of the test set is 100 s, the real-time factor is 0.004x.

The runtime measurements do not include the overhead for collecting the input data from the on-device microphone. As described in § V, OMG uses the capabilities from SANCTUARY to securely connect to sensors. Thus, only the world switch from an SA to the secure world to request the sensor data and the switch back to the SA introduce some overhead. As presented in [11], the switch from an SA to the secure world takes around 0.3 ms. Therefore, even in the short-running speech processing use case presented in this paper, the performance overhead introduced by reading sensor data via the secure world is negligible.

Our evaluation of a keyword recognition task using spectral fingerprints and a basic CNN lays the groundwork to port larger and recurrent architectures as well as to study training tasks. Since our implementation has no inherent memory limitations, it also allows to securely run more complex end-to-end systems, such as the recently released TensorFlow-based dictation model by Google [6], making it highly practical.

VII. Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 850990 PSOTI). It was supported by the DFG (HWSec, project A.1 within the RTG 2050 "Privacy and Trust for Mobile Users", and P3, S2, and E4 within CROSSING), by the BMBF and HMWK within CRISP, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

REFERENCES

[1] "Amazon Alexa User Receives 1,700 Audio Recordings of a Stranger through 'Human Error'," https://www.washingtonpost.com/technology/2018/12/20/amazon-alexa-user-receives-audio-recordings-stranger-through-human-error/, 2018.

[2] "Amazon Ordered to Give Alexa Evidence in Double Murder Case," https://www.independent.co.uk/life-style/gadgets-and-tech/news/amazon-echo-alexa-evidence-murder-case-a8633551.html, 2018.

[3] "Apple contractors 'regularly hear confidential details' on Siri recordings," https://www.theguardian.com/technology/2019/jul/26/apple-contractors-regularly-hear-confidential-details-on-siri-recordings, 2019.

[4] "Major breach found in biometrics system used by banks, UK police and defence firms," https://www.theguardian.com/technology/2019/aug/14/major-breach-found-in-biometrics-system-used-by-banks-uk-police-and-defence-firms, 2019.

[5] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang, Q. Liang, D. Bhatia, Y. Shangguan, B. Li, G. Pundak, K. C. Sim, T. Bagby, S. Chang, K. Rao, and A. Gruenstein, "Streaming End-to-end Speech Recognition for Mobile Devices," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019.

[6] J. Schalkwyk, "An All-Neural On-Device Speech Recognizer," https://ai.googleblog.com/2019/03/an-all-neural-on-device-speech.html, 2019.

[7] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel," in *USENIX Security*. USENIX, 2019.

[8] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership Inference Attacks Against Machine Learning Models," in *IEEE S&P*. IEEE, 2017.

[9] N. Carlini, C. Liu, J. Kos, Ú. Erlingsson, and D. Song, "The Secret Sharer: Measuring Unintended Neural Network Memorization & Extracting Secrets," *CoRR*, vol. abs/1802.08232, 2018.

[10] "TensorFlow Lite for Microcontrollers," https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/experimental/micro.

[11] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space Enclaves," in *NDSS*. Internet Society, 2019.

[12] B. D. Rouhani, H. Chen, and F. Koushanfar, "DeepSigns: An End-to-End Watermarking Framework for Ownership Protection of Deep Neural Networks," in *ASPLOS*. ACM, 2019.

[13] H. Chen, B. D. Rouhani, C. Fu, J. Zhao, and F. Koushanfar, "DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models," in *International Conference on Multimedia Retrieval (ICMR)*. ACM, 2019.

[14] C. Orlandi, A. Piva, and M. Barni, "Oblivious Neural Network Computing via Homomorphic Encryption," *EURASIP Journal on Information Security*, 2007.

[15] A.-R. Sadeghi and T. Schneider, "Generalized Universal Circuits for Secure Evaluation of Private Functions with Application to Data Classification," in *International Conference on Information Security and Cryptology (ICISC)*. Springer, 2008.

[16] M. Barni, P. Failla, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider, "Privacy-Preserving ECG Classification With Branching Programs and Neural Networks," *Trans. Information Forensics and Security (TIFS)*, vol. 6, no. 2, 2011.

[17] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *ICML*. JMLR, 2016.

[18] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *IEEE S&P*. IEEE, 2017.

[19] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious Neural Network Predictions via MiniONN Transformations," in *CCS*. ACM, 2017.

[20] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications," in *ASIACCS*. ACM, 2018.

[21] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," in *USENIX Security*. USENIX, 2018.

[22] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: XNOR-based Oblivious Deep Neural Network Inference," in *USENIX Security*. USENIX, 2019.

[23] T. van Elsloo, G. Patrini, and H. Ivey-Law, "SEALion: A Framework for Neural Network Inference on Encrypted Data," *CoRR*, vol. abs/1904.12840, 2019.

[24] M. Dahl, J. Mancuso, Y. Dupis, B. Decoste, M. Giraud, I. Livingstone, J. Patriquin, and G. Uhma, "Private Machine Learning in TensorFlow using Secure Computation," *CoRR*, vol. abs/1810.08130, 2018.

[25] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data," in *Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*, 2019, to appear.

[26] M. A. Pathak, B. Raj, S. Rane, and P. Smaragdis, "Privacy-Preserving Speech Processing: Cryptographic and String-Matching Frameworks Show Promise," *IEEE Signal Processing Magazine*, vol. 30, no. 2, 2013.

[27] F. Tramèr and D. Boneh, "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware," in *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2019.

[28] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *USENIX Security*. USENIX, 2016.

[29] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *IEEE S&P*. IEEE, 2015.

[30] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *IEEE S&P*. IEEE, 2015.

[31] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving Machine Learning as a Service," *CoRR*, vol. abs/1803.05961, 2018.

[32] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," *Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, 2018.

[33] N. Hynes, R. Cheng, and D. Song, "Efficient Deep Learning on Multi-Source Private Data," *CoRR*, vol. abs/1807.06689, 2018.

[34] S. Chandra, V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, and B. Thuraisingham, "Securing Data Analytics on SGX with Randomization," in *ESORICS*. Springer, 2017.

[35] F. Brasser, T. Frassetto, K. Riedhammer, A.-R. Sadeghi, T. Schneider, and C. Weinert, "VoiceGuard: Secure and Private Speech Processing," in *INTERSPEECH*. ISCA, 2018.

[36] S. Ahmed, A. R. Chowdhury, K. Fawaz, and P. Ramanathan, "Preech: A System for Privacy-Preserving Speech Transcription," *CoRR*, vol. abs/1909.04198, 2019.

[37] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz, "MLCapsule: Guarded Offline Deployment of Machine Learning as a Service," *CoRR*, vol. abs/1808.00590, 2018.

[38] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptology ePrint Archive*, vol. 2016/086, 2016.

[39] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-Process Memory Isolation EXtension," in *USENIX Security*. USENIX, 2018.

[40] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V," in *NDSS*. Internet Society, 2019.

[41] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *IEEE S&P*. IEEE, 2015.

[42] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation," in *IEEE S&P*. IEEE, 2010.

[43] "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, 2009.

[44] "Trust Issues: Exploiting TrustZone TEEs," https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017.

[45] "Zircon Microkernel," https://fuchsia.googlesource.com/zircon.

[46] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, 1978.

[47] P. Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition," *CoRR*, vol. abs/1804.03209, 2018.

[48] T. Sainath and C. Parada, "Convolutional Neural Networks for Small-Footprint Keyword Spotting," in *INTERSPEECH*. ISCA, 2015.

E

DARWIN: Survival of the Fittest Fuzzing Mutators (NDSS'23)

---

[97] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: Survival of the Fittest Fuzzing Mutators. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 24-27, 2023*, 2023. CORE Rank A*. Chapter 5.

DARWIN: Survival of the Fittest Fuzzing Mutators

Patrick Jauernig*, Domagoj Jakobovic‡, Stjepan Picek§, Emmanuel Stapf* and Ahmad-Reza Sadeghi†
*Technical University of Darmstadt, Germany, {patrick.jauernig, emmanuel.stapf}@sanctuary.dev
†Technical University of Darmstadt, Germany, ahmad.sadeghi@trust.tu-darmstadt.de
‡University of Zagreb, Croatia, domagoj.jakobovic@fer.hr
§Radboud University and TU Delft, The Netherlands, picek.stjepan@gmail.com

***Abstract*—Fuzzing is an automated software testing technique broadly adopted by the industry. A popular variant is mutation-based fuzzing, which discovers a large number of bugs in practice. While the research community has studied mutation-based fuzzing for years now, the algorithms' interactions within the fuzzer are highly complex and can, together with the randomness in every instance of a fuzzer, lead to unpredictable effects. Most efforts to improve this fragile interaction focused on optimizing seed scheduling. However, real-world results like Google's FuzzBench highlight that these approaches do not consistently show improvements in practice. Another approach to improve the fuzzing process algorithmically is optimizing mutation scheduling. Unfortunately, existing mutation scheduling approaches also failed to convince because of missing real-world improvements or too many user-controlled parameters whose configuration requires expert knowledge about the target program. This leaves the challenging problem of cleverly processing test cases and achieving a measurable improvement unsolved.**
We present DARWIN, a novel mutation scheduler and the first to show fuzzing improvements in a realistic scenario without the need to introduce additional user-configurable parameters, opening this approach to the broad fuzzing community. DARWIN uses an Evolution Strategy to systematically optimize and adapt the probability distribution of the mutation operators during fuzzing. We implemented a prototype based on the popular general-purpose fuzzer AFL. DARWIN significantly outperforms the state-of-the-art mutation scheduler and the AFL baseline in our own coverage experiment, in FuzzBench, and by finding 15 out of 21 bugs the fastest in the MAGMA benchmark. Finally, DARWIN found 20 unique bugs (including one novel bug), 66% more than AFL, in widely-used real-world applications.

I. Introduction

Vulnerabilities caused by programming errors are still a major threat to today's programs [47]. An important class of programming errors is memory corruption vulnerabilities, where unexpected, malformed inputs can lead to uncontrolled behavior in the program, which can often be abused by attackers. A modern, cost-efficient strategy to uncover these programming errors is automated software testing using fuzz testing (commonly known as *fuzzing*). Fuzzing automatically generates inputs from testcases and feeds them to the program under test while monitoring the program. If a programming error has been reached, the fuzzer notices that the program hangs or crashes. Optionally, the observed control-flow changes can serve as feedback for the next iteration, i.e., whether a new path in the control flow (known as *coverage*) has been taken due to the generated input. In recent years, fuzzers emerged as an important topic in academic as well as industrial research and are nowadays widely used for finding bugs in commercial software [41], [28]. Projects like Google OSSFuzz [28] helped to significantly increase the adoption rate by offering free computation for fuzzing while still allowing security researchers, who provide the fuzzers, to keep the bug bounty for discovered vulnerabilities.

While fuzzers are responsible for discovering tremendous amounts of bugs, even in operating system kernels [62], they are still extensively researched, e.g., in the areas of making targets available to fuzz testing [69], [20], improving fuzzers using new algorithms [7], [11], [38], [39], [37], [51], and leveraging new hardware features for performance or coverage improvements [56], [13].

This paper focuses on the subject of algorithmic improvements for mutational fuzzers, which leverage an existing set of testcases (referred to as *corpus*) to constantly generate new variants of these testcases by applying *mutation operators* inspired by genetic mutations. Most notably, a significant number of works focused on the effects of algorithmically sampling a subset of optimal seeds from the corpus. The goals of these works range from removing redundancy to creating a minimal coverage-preserving corpus with small files [67], [48], efficiently reaching specific locations in the control-flow graph [7], [11], [70], or improving coverage in general [44]. While these approaches are designed to select from a large number of possible testcases, in reality, testcases suitable for fuzzing are often rare [29].

Aside from seed-selection algorithms, other approaches have been proposed [37], [38], [51] that approximate which byte positions in the testcase give the best results when being mutated, but not which mutation operators to apply. Yet, this problem is highly challenging, as it is required to be shown whether 1) mutation selection is actually target-dependent, 2) the selection distribution is static or dynamic, 3) introducing an optimization algorithm reduces execution speed s.t. its better mutation selection is outweighed.

The first approach to optimize the actual selection of mutations (*mutation scheduling*) has been MOPT [39]. MOPT proposes a variant of the Particle Swarm Optimization algorithm (PSO) to learn a globally optimal mutation probability distribution. However, MOPT's PSO algorithm has both local and global best probability distributions, making finding the best solution, and therefore the algorithm itself, complex and more expensive to use during fuzzing. Similar to other algorithmic

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23159
www.ndss-symposium.org

improvements to fuzzing, finding a practical trade-off between complexity and algorithmic improvements is challenging. All additional algorithms have direct implications on execution speed, and hence, reduce coverage over time. Further, MOPT introduces various user-configurable parameters that steer the optimization process directly, so the user needs to solve another complex problem instead to avoid non-optimal scheduling. For a reasonable choice of parameters, the user either needs expert knowledge of the target application or a preliminary fuzzing campaign. Finally, MOPT fails to outperform AFL, which is built on, in the popular FuzzBench fuzzer benchmark by Google [27]. This makes designing and building a practical mutation scheduler a challenging open problem.

This work. This paper focuses on one aspect of the fuzzing process: finding (approx.) optimal mutation scheduling strategies to improve fuzzing algorithmically. Here, the challenging goal is to infer which mutation operator is the optimal choice for the next fuzzing iteration. We address this problem with DARWIN, a novel mutation-scheduling algorithm to improve the general performance of mutational fuzzers. DARWIN leverages an Evolution Strategy (ES), in a setting similar to reinforcement learning, to approximate ideal probability distribution for the mutation operator selection to avoid wasting fuzzing iterations on suboptimal mutators. The resulting probability distribution is not statically set but learned during the fuzzing process and dynamically adapted to the target program. DARWIN outperforms related work significantly, not only in coverage but also in the time to find bugs, without the user having to adjust any target-specific parameters, which allows non-expert users to leverage mutation scheduling.

Challenges. Although we focus only on a specific phase of the fuzzing process, namely the mutation selection in the havoc phase, the problem of finding an optimal probability distribution for mutation selection is highly challenging: numerous mutation operators can be used, and their efficiency varies depending on the target program, the current input, and the state inherently implied by the current input. Furthermore, the efficiency can vary depending on the non-deterministic nature of each fuzzing run and the interplay between fuzzing stages. Therefore, it is impossible to examine all possible options exhaustively in the general case.

Contributions. Our DARWIN mutation scheduler and its implementation based on AFL tackle all these challenges. To summarize, our main contributions include:

- We present a novel mutation scheduling approach, DARWIN; the first mutation scheduler that leverages a variant of Evolution Strategy to optimize the probability distribution of mutation operators. DARWIN dramatically improves the efficiency of mutation selection while keeping the execution speed constant. DARWIN can be applied to any feedback-guided mutation-based fuzzer.
- We implemented a prototype of DARWIN by extending AFL with our mutation scheduling algorithm. By modifying only three code lines in AFL to integrate our DARWIN mutation scheduler, we show that DARWIN's design is easily adoptable by existing fuzzers. We further highlight this by also integrating DARWIN in EcoFuzz[67]. What is more, we do not introduce any additional user-configurable parameters to avoid creating adoption barriers.
- We thoroughly evaluate DARWIN against AFL as a baseline and the most recent related work in this area, MOPT. Our prototype significantly outperforms both fuzzers, MOPT and AFL, in terms of code coverage reached in the well-fuzzed GNU binutils suite. Next, DARWIN is the first mutation scheduler to outperform its base fuzzer in Google's Fuzzbench. Further, we evaluate DARWIN on MAGMA, where we show that DARWIN triggers 15 out of the 21 bugs found the fastest. Finally, DARWIN finds 20 unique bugs (including one previously unreported bug), 66% more than AFL, across various real-world targets.
- We thoroughly analyze the root causes for DARWIN's efficiency by first comparing DARWIN to a static pre-optimized mutation probability distribution, and further, studying the mutation probability distribution over time, and introducing a metric to measure a fuzzer's effectiveness in scheduling mutations. We show that DARWIN needs fewer mutations than AFL to reach a coverage point while achieving a higher execution speed than the state-of-the-art MOPT fuzzer.

To foster future research in this area, we open-source our fuzzer at https://github.com/TUDA-SSL/DARWIN.

II. Background

This section presents the necessary background information to understand the general concept of fuzzers, the workflow of mutation-based fuzzers, and metaheuristic optimization.

A. Fuzzing

On a high level, fuzzers can be divided into mutational, i.e., mutating testcases, and generational, i.e., deriving structured inputs, fuzzers. Mutational fuzzing requires a set (*corpus*) of program inputs (*seeds*), which can, e.g., be obtained from testcases or real inputs. These seeds are then mutated using operations known from genetics, like inserting random errors (bit flips), changing values to corner cases, or combining two inputs to create a new input. As this way of input generation does not follow any constraints on the input, the generated inputs are more unlikely to pass, e.g., initial parser checks or checksums [63]. The process of mutation can be influenced in two ways: 1) the location in the input that gets mutated and 2) the mutation that is applied, whereby the selection can either be made randomly or guided by a heuristic. Such a heuristic can be, e.g., success measured in an increase of coverage or a certain state that should be reached (where the target has been tainted to find a clear path to that state). For example, the popular AFL fuzzer uses the coverage metric of basic-block transitions as a heuristic [29].

B. Fuzzing Loop of Mutational Fuzzers

For mutational fuzzers, the so-called fuzzing loop, which is the place in the code where the loaded seeds are mutated before being used as inputs for the program under test, usually can be divided into three stages, the deterministic, havoc, and splicing stage [56], [7], [11], [39], [4]. While some aspects are

AFL-specific, most concepts presented are implemented in a similar way for other fuzzers.

Deterministic stage. In the first stage, the deterministic stage, a small set of mutations is applied to seeds in a predefined order to create inputs for the target program, whereby the seeds are drawn from a queue of initial seeds provided by the user. AFL uses code coverage as a heuristic to decide whether a mutated seed has been *successful*. If a seed increases the code coverage, it is stored in the *fuzzing queue*. By reusing successful seeds in the later iterations, the overall fuzzing performance is improved. Measuring the code coverage is achieved by instrumenting the binary of the program under test such that the program is intercepted on every branch hit. When an input leads to a crash of the program under test, the user is notified since this indicates a bug in the program. The first stage of the fuzzing loop with its deterministic mutation scheme is slow and tends to contribute less to the overall coverage [39]. Thus, AFL allows disabling the deterministic stage entirely, which is especially beneficial for short fuzzing runs [39] or to reduce noise in performance measurements of the following stages.

Havoc stage. In the second stage of the fuzzing loop, the non-deterministic havoc stage, randomly chosen mutations are selected from a list of mutational operators [57], [52], [29], [12]. In Table X, Appendix E, we list the mutations used in AFL's havoc stage. The selected mutations are applied to the inputs received from the deterministic stage or to the mutated seeds from the fuzzing queue. When the generated program inputs achieve new coverage, they are again saved in the fuzzing queue. The fuzzing loop then returns to the deterministic stage and selects the next element from the fuzzing queue for the next iteration. The havoc stage is the most generic stage and widely adopted by AFL-based and other mutational fuzzers [52], [12], [29], [21], which is also why our novel mutation scheduler DARWIN targets the havoc stage.

Splicing stage. The last stage of the fuzzing loop, the splicing stage, is only activated when none of the inputs in the fuzzing queue led to new coverage in the havoc stage. In the splicing stage, a crossover mutation of two inputs is performed, which is then fed back to the havoc stage, which again applies a random mutation on the input before testing it on the target program.

C. Metaheuristics

While in the previous section, we mentioned several approaches to fuzzing, we did not discuss how such approaches can actually find good solutions. This is because there exist no specialized algorithms developed for that particular problem. Instead, we need to rely on more general solving procedures. Metaheuristics represent an intuitive choice since they encompass problem-independent techniques used in a broad range of applications. For example, we can consider the problem of finding a suitable *mutation schedule* in the havoc stage as an optimization problem. Since there is no explicit cost function for this optimization problem, it cannot readily be paired with classical optimization algorithms requiring gradient information. In that case, metaheuristic algorithms, which do not pose any requirements on the optimization problem, have proven to be the method of choice in many engineering applications. Metaheuristic techniques are commonly used in domains like the automotive industry [22], medicine [1], scheduling [9], adversarial examples [59], and implementation attacks [65].

Metaheuristics, in their original definition, represent solution finding methods that orchestrate an interaction between local improvement and higher-level strategies to create a process capable of escaping from local optima and performing a robust search in a solution space [26]. A common division of metaheuristic optimization algorithms is into single solution-based and population-based metaheuristics [60]. Population-based metaheuristics work on a population of solutions (e.g., Evolutionary Algorithms (EA) and swarm algorithms like Particle Swarm Optimization (PSO)). A population in this context denotes a set of individuals used during an optimization process, whereby an individual is a data structure that corresponds to an element in the search space (a candidate solution). In contrast, single solution-based metaheuristics manipulate and transform a single solution (or a smaller number of solutions) during the search.

Evolutionary algorithms occupy a prominent place among metaheuristic algorithms, as they have been successfully applied to a large number of difficult optimization problems [24], [54]. We depict pseudocode for the generic evolutionary algorithm in Algorithm 3, Appendix A. In each iteration, the algorithm applies a selection mechanism that emulates natural selection. Based on their respective quality, usually denoted as *fitness*, better individuals survive, while worse ones are eliminated. The population then undergoes variation, creating new genetic material as new individuals in the population. Finally, all the individuals are reevaluated, and the process is repeated until a specific termination criterion is met. Since no knowledge is presumed about the nature of the solutions in the current population, the termination is usually based on the number of iterations, allotted time, or finding a solution of acceptable quality.

Metaheuristic optimization algorithms balance diversification and intensification properties; diversification enables the discovery of promising areas in the search space and escaping from local optima. Intensification aims to exploit a promising area by concentrating on the current best solution and finding better neighboring solutions. The interplay of these properties determines the effectiveness of metaheuristic methods when applied to a specific optimization problem.

III. Challenges

Designing a mutation scheduling algorithm comes with a number of challenges, as mutation scheduling is a fragile part in the fuzzing process. These challenges are:

C.1: Optimal Mutation Selection. Finding an optimal probability distribution for mutation selection is challenging, as the optimal distribution might change per target. Further, the probability distribution might depend on the state implied by a part of the input (that is not mutated). Hence, a mutation scheduler needs to show that this mutation selection indeed needs to adapt dynamically and, if so, show that iterative adaption outperforms random selection.

C.2: Integrating an Optimization Algorithm. Properly selecting a candidate algorithm for mutation scheduling is itself

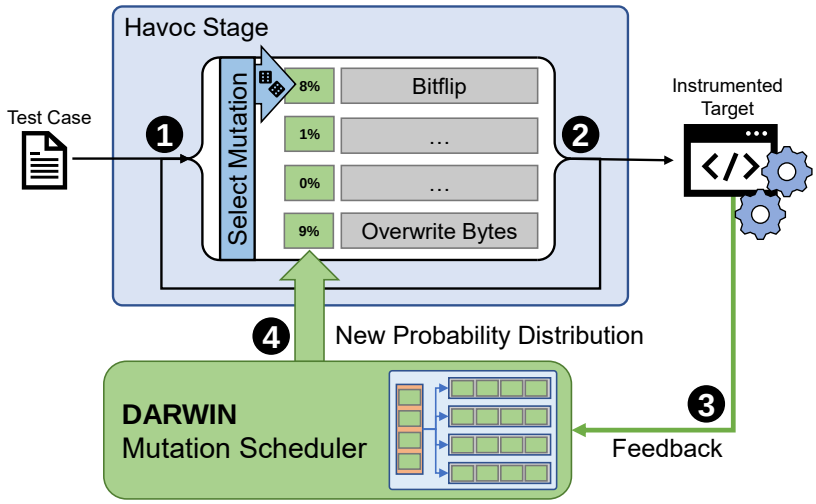


Fig. 1. High-level overview showing how DARWIN iteratively optimizes the probability distribution for mutation selection and how the selected mutations are applied to the testcases.

highly challenging. However, integrating this algorithm into the existing fuzzing process requires a 1) carefully designed representation not only of the problem but also the solution to avoid spending too much computation on encoding, 2) finding a parameter fit for the respective algorithm that fine-tunes exploration versus intensification.

C.3: Easy Adoption and Reproducibility. A complex approach with a large number of user-tweakable parameters might achieve outstanding results. However, it will still not be used in practice due to the difficulties in integrating the approach into a fuzzer or because users fail to find good parameter values, and hence, they cannot achieve results similar to the ones reported by the authors.

C.4: Performance Trade-off. Achieving an optimal trade-off for the mutation selection scenario, which is our goal, is complex. For instance, fuzzing approaches typically tune the trade-off between performance and cleverness in seed selection. Better seeds reach basic blocks guarded by complex constraints, but optimizing seed selection with algorithms takes additional time, and hence, decreases execution speed.

We designed DARWIN with these challenges in mind. Next, we explain how we addressed these challenges throughout the design, implementation, and evaluation of DARWIN.

IV. DARWIN Design

DARWIN is a novel mutation scheduling algorithm using an Evolution Strategy (ES) to find an optimal mutation selection probability distribution to be applied during the havoc stage. DARWIN is not only determining a static probability distribution but keeps on adapting the distribution throughout the fuzzing run based on coverage information. Our approach, as depicted in Figure 1, comprises a well-defined optimization module that does not need to expose any parameters to the user of the fuzzer. In detail, a fuzzer featuring DARWIN performs the following steps in the havoc stage (each step is marked in Figure 1):

1) At the beginning of the havoc stage, the fuzzer selects an input from the queue and randomly selects the next mutation to apply. Initially, the probability distribution for mutation selection is uniform.
2) After applying a mutation, the fuzzer decides whether it should keep mutating this input or if the input should be tested on the instrumented application.
3) After running the instrumented application with the selected input, feedback is reported to assign a success score to the test input and the DARWIN Mutation Scheduler. The mutation scheduler learns based on the reported feedback and optimizes the probability distribution using DARWIN's Evolution Strategy.
4) Finally, the updated probability distribution is applied for the next iteration.

In the following, we explain the optimization process of DARWIN's Mutation Scheduler in more detail.

A. Metaheuristics and Mutation Scheduling

In the context of the complete fuzzing pipeline, we concentrate on improving the mutation scheduler, as illustrated in Figure 1. The problem of finding a suitable mutation schedule is considered here as an optimization problem, where the candidate solution is a vector of relative mutation operator probabilities. In a classical optimization scenario, a candidate solution is refined through a series of iterations. In each iteration, the candidate is evaluated, which is usually the most time-consuming part of the optimization. Only after a number of iterations, when a candidate of acceptable quality is obtained, the solution is applied to the process being optimized.

In the case of fuzzing, however, the optimization is performed concurrently with the process being optimized since each candidate solution is used as it is being evaluated, and the optimization is performed for each target independently. Because of this, the optimization algorithm should be able to provide a fast convergence, which means as large a performance improvement with as few evaluations as possible.

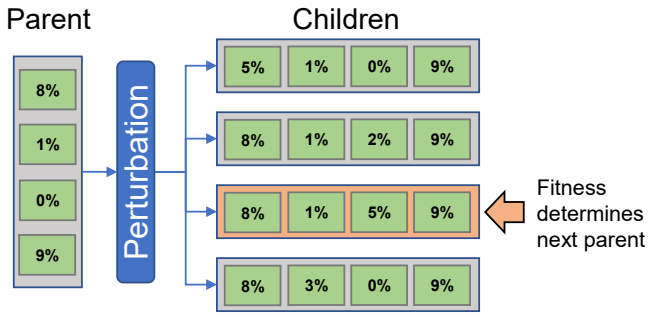


Fig. 2. Example of an ES instantiation with one parent and four children ($\mu = 1, \lambda = 4$). Based on the fitness function, the parent for the next iteration is determined.

As mentioned in Section II-C, metaheuristic techniques balance between diversification and intensification, with conflicting goals to evade local optima and, at the same time, enable convergence to better quality solutions. Population-based metaheuristics, such as Genetic Algorithm (GA) [43] or Particle Swarm Optimization (PSO) [55] are generally focused on diversification and can locate an optimum with a greater probability. However, as mentioned above, the optimum in the fuzzing process is not fixed, and the algorithm should adapt swiftly to the current target. Since they need to evaluate a population of candidate solutions in every step, these approaches usually require a large number of evaluations, and consequently, computation time, to reach a solution of acceptable quality. Those methods may also include computationally intense domain-dependent operators acting on multiple solutions, such as the crossover operator in GAs, which is a process where a new individual is created from two or more parent solutions [16]. Since, in our case, fast convergence and ease of use are the primary goals, population-based metaheuristics do not present an appropriate choice.

Instead of population-based methods, algorithms that operate on a single solution (or a small set of solutions) should prove to be a better option. It is expected that single solution algorithms will obtain better performance; since they primarily focus on intensification, convergence is usually faster than in the population-based methods [45].

In optimizing fuzzing mutation probabilities, where each evaluation may take a considerable amount of time, this behavior translates into a far smaller number of evaluations needed to reach an acceptable solution quality. At the same time, such algorithms still provide a means to escape local optima with solution perturbations and random restarts. Examples of these algorithms include Simulated Annealing (SA) [36], Tabu Search (TS) [25], and Evolution Strategy (ES) [6].

B. *Evolution Strategy as used in* DARWIN

When considering domain-independent optimization methods, as is the case here, Evolution Strategy has proven to be an efficient and versatile method found in a multitude of applications [17], [23]. As such, we opted to use ES as the method of choice, both for its simplicity and proven track record as a multi-purpose optimization algorithm [6]. Additionally, ES is well-known to be robust [5], making it an ideal choice when dealing with difficult optimization problems.

The intensification process in metaheuristics is commonly performed with the use of a mutation operator. Mutation operators use only one parent and create one child by applying a randomized change to its genotype (i.e., the encoding of an object) [16]. However, since we already use the term "mutation" for changes in seeds performed by the fuzzer, we will slightly bend the terminology and denote the mutation operator used in ES as the *perturbation* operator. We depict the process in Figure 2.

In its most common form, ES operates on a single solution μ, called the parent. In each iteration, a randomized perturbation operator is applied on the parent solution producing a number of different modified solutions, commonly called children. The number of children solutions is denoted with λ, which is a parameter of the algorithm. After every child solution is evaluated, the best among all the children solutions and the current parent is chosen as the parent in the next iteration. This allows DARWIN to adjust the mutation schedule dynamically, addressing Challenge C.1. This type of Evolution Strategy is denoted as $(\mu+\lambda)-ES$. If the parent is disregarded, such selection method is denoted as $(\mu, \lambda)-ES$. The process is repeated until a designated termination criterion is met, commonly based on elapsed time or a number of evaluations. We provide the ES pseudocode in Algorithm 1.

Algorithm 1 Evolution Strategy

```
1: initialize the parent solution
2: repeat
3:     create λ child solutions using perturbation on the parent
4:     select the best solution
5:     set the best solution as the parent
6: until TerminationCriterion
```

When using a single starting parent solution, the algorithm will mainly concentrate on its relative vicinity in the search space. While it is possible for the perturbation operator to move the search to a more distant area, this occurs with a lower probability. To allow the fuzzer to discover more promising areas in the search space (e.g., more efficient mutation operator combinations), DARWIN uses an extended form of the algorithm that starts not with one but several different starting parent solutions. In this case, the search is conducted in parallel, independently for each parent, addressing Challenge C.4. The number of parent solutions in this algorithm variant is denoted with μ. The modified algorithm can be represented with the following pseudocode:

Algorithm 2 Multi-parent Evolution Strategy

```
1: initialize μ parent solutions
2: repeat
3:     for all parent solutions do
4:         create λ child solutions using perturbation on the parent
5:         select the best solution
6:         set the best solution as the parent
7:     end for
8: until TerminationCriterion
```

Using algorithm parameters μ and λ, we can balance between the diversification and intensification segments of the

search. In our experiments, we have used the value of 4 for the parameter λ, which is a common choice in diverse ES applications, see, e.g., [33], [35], [42]. With this parameter value, the parent for the next iteration is selected among five solutions in total (the parent and four child solutions). If multiple parents are used, we set the parameter μ to the value of 5. Note that we experimented with several values for μ, and the main difference from the performance perspective is in the speed of convergence, realizing in slightly worse coverage in our preliminary experiments (cf. Section VI-B).

Let us consider more why taking a small λ size (but larger than 1) makes sense. First, if we consider an extreme case where λ equals 1, we effectively reach a local search algorithm. While such an algorithm could work for this problem, it would face issues with a high probability of getting stuck in local optima. The second extreme for λ would represent a large population size (e.g., order of magnitude 100). Then, we face two issues:

- Due to the large population size, we must conduct more evaluations [1], which will be a problem as fitness evaluation is computationally expensive.
- When having a large population size, it is also common to use the crossover operator to foster search space exploitation, which increases the computational complexity of the algorithm but also makes tuning more difficult. Indeed, by adding crossover, we must tune the algorithm for different crossover operators and the probability of the crossover action.

Finally, unlike in a classical optimization scenario, here, the algorithm's efficiency is not measured based just on the final, best solution the algorithm has found. Since the optimization is performed concurrently with the fuzzing, every candidate solution that appears during the algorithm run contributes to the overall fuzzing efficiency. For that reason, and because the optimization is performed per-target basis, it is important to provide a fast convergence, which can be acquired with a smaller population size.

C. Solution Encoding and Perturbation

The ES algorithm can be used with any form of solution encoding, as long as a suitable perturbation operator (or operators) is defined. In the case of optimizing the fuzzing mutation schedule, we used two solution encodings and corresponding perturbation operators.

First, we investigated an encoding that uses a *real-valued vector* to represent relative probabilities of mutation operators; this representation is equal to the one used in MOPT [39]. The size of the vector is equal to the number of mutation operators since each element of the vector represents the relative probability that a certain mutation operator (given in Table X) will be selected. In each invocation, the values in the vector are used to determine the next mutation operator. Initial values of vector elements are generated uniformly at random in the range $[0, 1]$. As the perturbation operator, we use a simple Gaussian perturbation with zero mean and standard deviation of 0.25; the obtained random value is added to a single randomly selected element in the vector (Figure 8(a), Appendix A). The value of 0.25 is selected after tuning, where we followed common reasoning for ES: the operator needs to be able to do significant changes (thus, we do not select a very small standard deviation), but it also should not behave like a random search (which would happen with a large standard deviation value). The values are always kept greater than zero but are allowed to exceed 1 (to allow the algorithm to emphasize an operator if needed).

The second encoding uses a *binary vector* (with values assuming only 0 and 1), where each element in the vector corresponds to a mutation operator. This simplifies the mutation operator choice so that only a subset of operators, whose corresponding values in the vector are 1, are used for mutation selection; among the elements of this subset, a random mutation is selected by the fuzzer. As the perturbation operator, a simple one-bit flip is used; each time a solution needs to be modified, a randomly selected bit in the vector is inverted (Figure 8(b), Appendix A).

We decided on the binary encoding for the solution encoding since a preliminary evaluation showed a geometric mean coverage increase of around 3%. What is more, with the binary encoding, we do not need to tune the standard deviation value for the perturbation operator (as we needed for the real-valued representation). This design decision addresses Challenge C.2.

D. Objective Function

The algorithms described above can be used with any conceivable performance measure related to the process being optimized. In this case, the primary criterion used for the evaluation of individual solutions is the number of unique paths encountered in the instrumented application. Unique paths encode all different ways to reach every possible basic block. While keeping track of all of them is tough (and leads to state explosion), counting new unique paths per iteration is simple and efficient. Hence, we decided to leverage the number of new unique paths as a feedback signal, especially since most fuzzers already provide this number. The solution with the highest number of paths will get selected as the next parent. Thus, our goal is the maximization of the following expression, which is in the evolutionary computation field commonly denoted as the fitness function:

$$fitness = \#\ Unique\_Paths \tag{1}$$

This performance measure follows previous work [39], but the proposed optimization method can be used to optimize a different criterion if necessary. An alternative approach to a single criterion would be to use a multi-objective optimization algorithm, but this choice is justified only when conflicting objectives need to be optimized concurrently, which is not the case here. Furthermore, using simpler fitness functions has the advantage of better interpretability, i.e., it is clear why a certain solution is better than some other one.

By combining our simple algorithm design (small population, no need for the user to tweak the parameters), support for various solution's encodings, and fitness function, we address Challenge C.3. We emphasize that Evolution Strategy

[1] Alternatively, we would need to reach good solutions in only a few generations, which is highly unlikely for a problem of such difficulty and the lack of structure in the genotype.

is commonly used in the $(\mu + \lambda)$ form, where standard values are 1 (note that here we talk about the number of parents in a single search, and not the total number of parents due to the parallel execution of ES) and 4, see, e.g., [6], [31]. Thus, while one could experiment with other values and then consider μ and λ as parameters that need to be tuned, our investigation shows this is unnecessary. Consequently, we do not consider μ and λ as user parameters, nor would the change of those values result in significant performance differences.

V. Implementation

We implemented a prototype of DARWIN in C as an extension to AFL 2.54b [29], a popular generic fuzzer that is leveraged by many research works as a foundation [39], [7], [4], [56]. DARWIN consists of about 320 lines of code. AFL is easily extendable and does not contain other algorithmic improvements itself, unlike projects like AFL++ [21] that try to incorporate all state-of-the-art improvements for best results in practice. For our DARWIN mutation scheduling algorithm, we added an interface to AFL to report feedback in the form of newly discovered paths from the instrumented application to the mutation scheduler. The interface exposes three functions: initialization, selecting a mutation, and reporting feedback to DARWIN. This enables a modular design for different mutation scheduling algorithms.

To derive the random numbers needed for our Evolution Strategy, we leverage the RomuDuoJr random number generator (RNG) [46] to balance out the higher reliance on the random number generation of DARWIN's ES algorithm. In Appendix B, we show that the speed difference is negligible compared to the standard RNG.

VI. Evaluation

We analyze DARWIN regarding a variety of aspects. First, we evaluate DARWIN's general ability to explore programs as an approximation for the fuzzer's efficiency in Section VI-A. Second, we evaluate the fuzzers in terms of execution speed in Section VI-C to ensure our efficiency improvement can be attributed to the novel mutation scheduling algorithm and that the algorithm does not have grave consequences on execution speed. Finally, we evaluate DARWIN's ability to find crashes using the LAVA-M [14] (Appendix D) and MAGMA [32] (Section VI-D) benchmarks, to show that the aforementioned aspects lead to finding more bugs faster.

Setup. Our evaluation setup across all experiments consists of four workstations with an AMD EPYC 7402P 24-Core processor and 256GB of RAM (to perform the evaluation in parallel while keeping memory accesses independent). The target applications, fuzzers, and seeds are all stored on a ramdisk to reduce the influence of disk I/O. Each evaluation run is executed sequentially on a dedicated machine to reduce the influence of, e.g., memory bandwidth.

We evaluate DARWIN against the most-related work, MOPT, and AFL 2.54b as a baseline (as both DARWIN and MOPT extend AFL). We ported MOPT to AFL 2.54b (by diffing AFL 2.52b and AFL 2.54b) to ensure that MOPT got the same bug fixes that DARWIN and AFL have.

TABLE I. Invocation of benchmark tools and file formats used as seeds.

Benchmark	Invocation	Format
bsdtar	-xf @@ /dev/null	TAR
cxxfilt	-t	ELF
djpeg	@@	JPEG
jhead	@@	JPEG
objcopy	–dump-section text=/dev/null @@ /dev/null	ELF
objdump	-d @@	ELF
readelf	-a @@	ELF
size	@@	ELF
strip	-o /dev/null @@	ELF
tcpdump	-nr @@	PCAP

Evaluation of fuzzers is, as with most research topics in security, not standardized, leading to fluctuating results reported in papers and varying results in practice. This is mainly due to two aspects: 1) evaluating related work with non-optimal parameters and 2) missing statistical analysis of the results. For the former, we disable the deterministic stage of DARWIN and AFL for all experiments completely while using the corresponding Pacemaker mode (with the parameter "-L 0") to achieve the same effect and focus on the havoc stage for MOPT. Note that this is crucial for a fair comparison [66]. For the latter, we integrated the approaches proposed by Klees et al. [34] to the best of our knowledge and investigated broadly used fuzzing benchmarks to reason about DARWIN's performance.

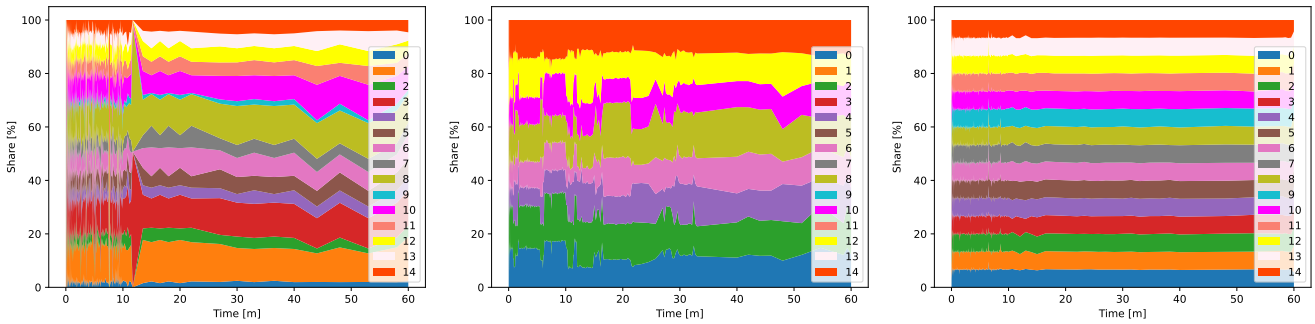
A. Evaluating Coverage

In the first step, we use code coverage as a proxy metric for a fuzzer's success. While code coverage is a well-established quality measure in related work [39], [67], [7], [4], it merely approximates the fuzzer's capabilities in finding bugs, as a fuzzer needs to cover a line of code to find a bug in it.

In all experiments, we leverage six applications, which process an executable ELF file without modifying it, from the well-fuzzed GNU binutils suite [2] in version 2.34 [4], [67], [39], [37]. We further include jhead 3.06.0.1, bsdtar (from libarchive) 3.6.0, tcpdump 4.99.1, and djpeg 2.1.2, as they are also commonly used [67], [4], [39], [37]. For increased reproducibility, we also kept the number of seed files low. Otherwise, as each seed is selected randomly by default, the variance for each run increases. The seeds used for the binutils targets always remain the same: one uninformed, empty test case and one minimal correct test case. We used the standard testcases bundled with AFL, except for binutils, where we used a minimal C program (smaller than the one bundled) described in Appendix C.

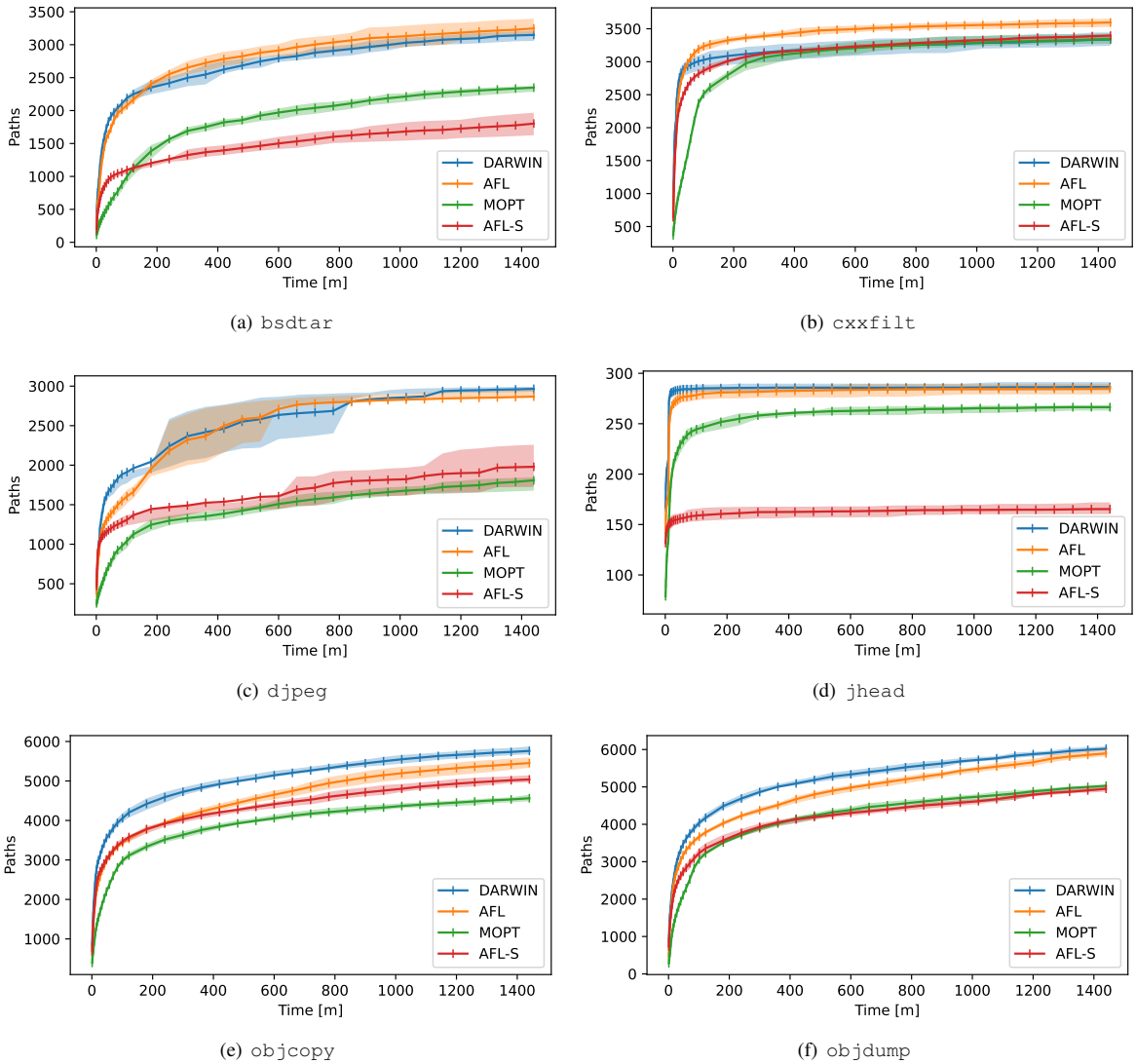
We evaluate the performance of the selected fuzzers over three independent runs, reporting the mean and 25%/75% quartiles. Each experiment runs for 24 hours. We present the mean coverage for each benchmark but also the standard deviation over time. Further, we additionally conduct the non-parametric Mann-Whitney U test to evaluate whether there are statistically significant differences among results, as suggested by Arcuri et al. [2] and Klees et al. [34].

[2]https://www.gnu.org/software/binutils/



(a) DARWIN (b) MOPT (c) AFL

Fig. 3. Mutation history for `cxxfilt`.



(a) `bsdtar` (b) `cxxfilt`

(c) `djpeg` (d) `jhead`

(e) `objcopy` (f) `objdump`

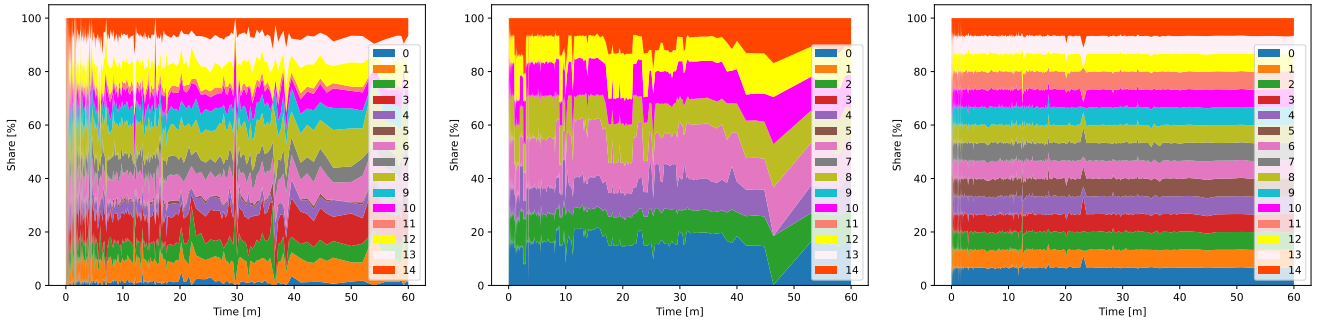
Fig. 4. The coverage results on the various benchmarks for AFL, MOPT, DARWIN, and the statically optimized variant AFL-S. Shaded areas represent the respective 25%/75% quartiles.

The results of our coverage evaluation for DARWIN, MOPT, and AFL are depicted in Table II. In Figure 4 and Figure 6, we show the respective graphs for coverage over time.

First of all, we can observe that MOPT is constantly performing worse than DARWIN, as well as AFL (except in one experiment). For `djpeg`, `jhead`, `objcopy`, `objdump`,

TABLE II. MEAN COVERAGE RESULTS MEASURED IN UNIQUE PATHS AND EDGES FOR WELL-FUZZED TARGETS OVER TEN RUNS. AFL-S IS AFL WITH OPTIMIZED, STATIC PROBABILITY DISTRIBUTION. GEOMETRIC MEAN IMPROVEMENT ("GEOMEAN") OF DARWIN OVER MOPT AND AFL, RESPECTIVELY. P-VALUES FOR THE MANN-WHITNEY U TEST FOR DARWIN ON THE NUMBER OF UNIQUE PATHS FOUND IN 24H. P-VALUES FOR THE MANN-WHITNEY U TEST FOR DARWIN ON THE NUMBER OF UNIQUE PATHS FOUND IN 24H. NOTE THAT EXPERIMENTS WITH A SIMILAR RESULT ACROSS SAMPLES (ITALIC) LEAD TO A HIGH P-VALUE NATURALLY; ALL REMAINING EXPERIMENTS ARE STATISTICALLY SIGNIFICANT WITH $p < 0.05$.

	DARWIN		MOPT			AFL			AFL-S		
Benchmark	unique paths	edges	unique paths	edges	p-value	unique paths	edges	p-value	unique paths	edges	p-value
`bsdtar`	3147.20	**5369.70**	2347.50	4832.0	9.13e-05	**3246.50**	5302.60	*0.093*	1801.30	4970.90	9.08e-05
`cxxfilt`	3334.18	2327.27	3343.00	2333.09	0.0001	**3594.91**	2425.36	1.95e-04	3395.50	**2500.30**	*0.647*
`djpeg`	**2964.60**	**3191.00**	1807.80	2765.90	9.13e-05	2866.00	3148.80	*0.163*	1978.50	2851.80	9.13e-05
`jhead`	**285.40**	**340.00**	265.4	339.00	2.17e-04	283.90	**340.00**	*0.520*	164.30	336.00	8.88e-05
`objcopy`	**5760.82**	**7912.36**	4562.00	7606.00	4.08e-05	5453.09	7881.27	4.05e-04	5038.20	7507.90	6.20e-05
`objdump`	**6018.91**	**7269.82**	5028.82	7003.73	4.06e-05	5895.91	7141.55	0.028	4947.90	7044.00	6.20e-05
`readelf`	**29715.64**	13012.36	26686.73	12273.00	4.08e-05	29439.27	12032.18	*0.162*	29519.90	**13019.20**	*0.805*
`size`	**3020.91**	**4030.91**	2206.82	3773.45	4.07e-05	2726.91	3809.55	5.32e-05	2861.50	3941.00	8.24e-04
`strip`	**5732.55**	7703.55	4497.36	7470.82	4.08e-05	5519.36	**7756.45**	0.001	5047.60	7354.30	6.20e-05
`tcpdump`	**9361.20**	**13834.10**	4723.60	11618.70	9.13e-05	9354.10	13317.2	*1.0*	4255.70	11952.10	9.13e-05
geomean			+29.40%	+6.77%		+1.60%	+1.73%		+32.35%	+4.38%	



(a) DARWIN (b) MOPT (c) AFL

Fig. 5. Mutation history for `size`.

`size`, and `tcpdump`, DARWIN clearly reaches the highest number of paths and edges, and also has the steepest increase in unique paths found over time for the first hour of fuzzing. For `objcopy` and `strip`, we saw that DARWIN the probability for mutation 0 (flip single bit) and 14 (overwrite bytes with a randomly selected chunk) tremendously, whereas DARWIN reduces probability of mutation 4 (randomly subtract from byte) for `objdump` and 5 (randomly add to byte) for `size`. Besides looking only at the paths covered, we can also consider the time to the same coverage as a figure of merit. For example, for `size`, DARWIN reaches AFL's maximum coverage approximately 800 minutes earlier, similar for `objcopy` and `objdump` where DARWIN reaches the same point approx. 700 minutes earlier.

The `cxxfilt` benchmark shows such a different behavior than other benchmarks that it warrants further discussion. This is the only case where AFL is a clear winner, and both mutation-scheduling-based fuzzers reach a similar coverage. While we noticed that AFL is achieving new coverage with the splicing stage around 50% more often than DARWIN, MOPT found four times as many coverage-triggering inputs using splicing. As such, we can exclude splicing being one reason for this effect.

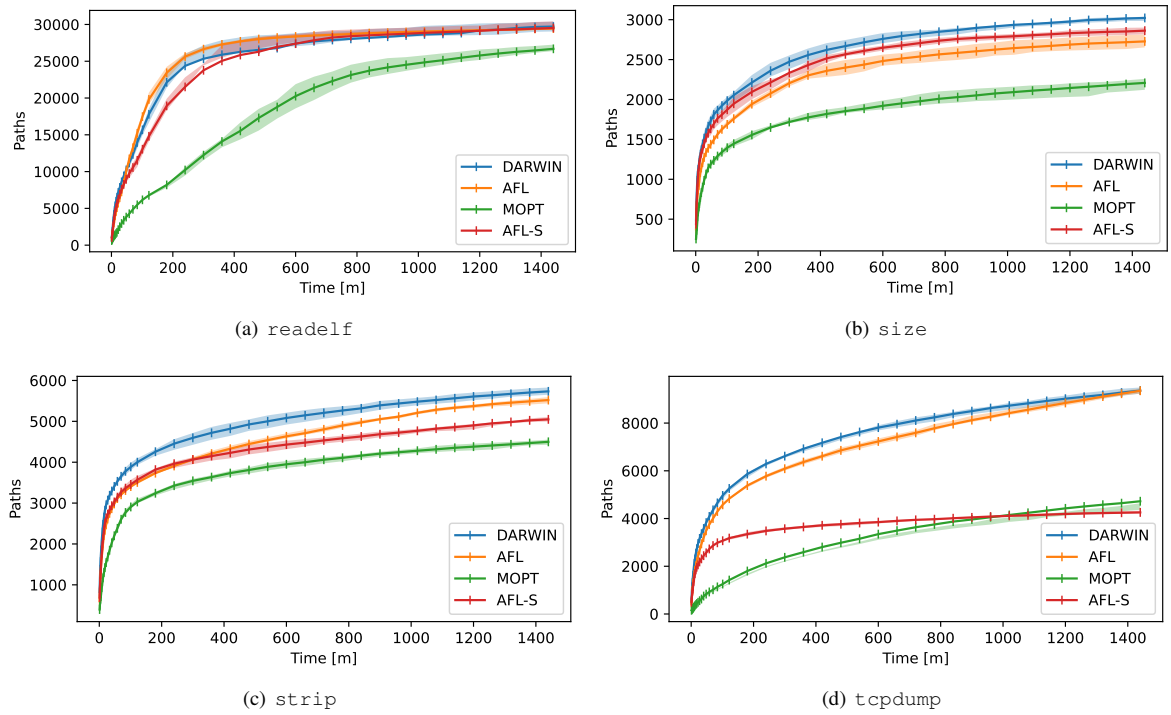
Hence, we looked at the mutations scheduled within a timespan of 1h, as shown in Figure 3. There we can see that DARWIN as well as MOPT put more and more emphasis on mutators 8 and 10 after around 40 minutes. This is also the very same moment where AFL starts to outperform both fuzzers. As `cxxfilt` is aiming at demangling overloaded functions (and the similarly behaving bsdtar is unpacking archives), it seems like mutation schedulers only add little benefit to fuzzing targets that are heavily relying on parsing. Yet, their performance impact (as we explore later) reduces the raw execution speed of the fuzzer, resulting in inferior coverage results.

Looking at `size` Figure 5, a target where DARWIN significantly outperforms AFL and MOPT, DARWIN avoids scheduling mutators 0, 5, 11, while mutator 0 has a large share in MOPT.

While analyzing the mutation histories, we noticed that MOPT schedules only 8 of the 15 mutations across all of our benchmarks. Most likely, this is an implementation bug as there is no visible calibration effect (in comparison to, e.g., the first 10 minutes of DARWIN, where DARWIN converges quickly afterward). This is also one possible factor for the diverse results MOPT shows in our experiments.

In conclusion, DARWIN shows a geometric mean improvement in edge coverage of 6.77% over MOPT, and 1.73% over AFL, hence, this addresses Challenge C.1. While this might seem insignificant at first, coverage measurements are only an approximation of a fuzzer's efficiency in finding bugs, as we show later.

FuzzBench. FuzzBench [40] is a fuzzing benchmark suite developed by Google. The benchmark comprises various widely-fuzzed real-world targets, e.g., from OSS-Fuzz [28]. We conducted a local FuzzBench coverage experiment over ten runs, where each run took six hours. All Fuzzbench experiments



(a) `readelf` (b) `size`

(c) `strip` (d) `tcpdump`

Fig. 6. The coverage results on various benchmarks for AFL, MOPT, DARWIN, and the statically optimized variant AFL-S. Shaded areas represent the respective 25%/75% quartiles.

TABLE III. MEDIAN RELATIVE CODE-COVERAGES ON EACH BENCHMARK AFTER 10 RUNS WITH 6H EACH. MEDIAN RELATIVE PERFORMANCE OF EACH FUZZER TO THE ENCOUNTERED EXPERIMENT MAXIMUM.

	DARWIN	AFL	MOPT
FuzzerMedian	**97.11**	96.89	86.70
FuzzerMean	**96.34**	95.46	83.64
bloaty_fuzz_target	**96.40**	94.95	89.62
curl_curl_fuzzer_http	**98.35**	97.25	92.19
freetype2-2017	**94.80**	93.68	78.74
harfbuzz-1.3.2	**98.95**	97.70	86.48
libjpeg-turbo-07-2017	**88.81**	88.72	69.20
libpng-1.2.56	**99.72**	98.79	94.05
libxml2-v2.9.2	93.34	**96.89**	61.79
libxslt_xpath	**97.11**	92.21	83.69
mbedtls_fuzz_dtlsclient	**98.94**	97.63	95.26
openssl_x509	**99.87**	**99.88**	99.73
openthread-2019-12-23	88.74	**88.84**	86.70
php_php-fuzz-parser	96.78	**98.94**	94.64
proj4-2017-08-14	**94.95**	93.40	28.57
re2-2014-12-09	**98.45**	98.34	83.51
sqlite3_ossfuzz	**92.44**	86.38	78.14
systemd_fuzz-link-parser	**99.92**	99.84	97.97
vorbis-2017-12-11	**97.01**	96.77	84.87
woff2-2016-05-06	**97.78**	95.75	91.88
zlib_zlib_uncompress_fuzzer	**98.12**	97.71	92.14

were conducted on a workstation with an Intel Xeon Silver 4110 CPU with 2.10GHz and 128GB RAM.

Experiments are depicted in Table III. DARWIN outperforms both AFL and MOPT in the avg. normalized score and avg. rank. Specifically, DARWIN reaches the highest median relative code coverage in 15 out of 19 experiments, is even with AFL in two (DARWIN has in `openssl_x509` 0.00001% and in `openthread-2019-12-23` -0.11% less coverage).

In the remaining two experiments, AFL slightly outperforms DARWIN: `libxml2-v2.9.2` (3.80%) and `php_php-fuzz-parser` (2.23%) are both parsers, as such, coverage mainly comes from well-structured testcases. As DARWIN does not improve testcase generation itself, e.g., using grammars, both fuzzers generate testcases of similar (bad) quality and hence, largely fail to cover a big part of the targets. AFL's faster execution speed allows it to generate more testcases per second, which is the cause for the differences.

MOPT is last in every experiment, with `openssl_x509` being the experiment closest to DARWIN and AFL. Thus, DARWIN is the first mutation scheduler to show coverage improvements over AFL in FuzzBench.

Static Optimization vs. Adaptive Optimization. For the mutation scheduling problem at hand, it is not clear if the perfect mutation probability distribution changes over time with the same target application. Hence, we used DARWIN to fuzz the targets from Section VI-A for 24h, but this time, storing the "best so far" parent in the current set of parents after 24h. As shown in Table II, DARWIN outperforms the static variant (referred to as AFL-S) by 4.38% geometric mean in the number of covered edges (and 32.35% in paths). Especially in the non-binutils experiments, DARWIN shows the importance of adaptive optimization throughout the fuzzing process.

In binutils, the static variant is much closer to the adaptive variant,as a lot of library code is shared between the individual applications, and the inputs are always executables. This also reflects in the resulting probability distributions, i.e., `readelf`, `size`, and `cxxfilt` share the same distribution,

and `strip`, `objcopy`, and `objdump` share the same distribution. Both groups have 7 disabled mutations and commonly disable mutations 3, 5, and 8 (cf. Appendix B). From our investigations, the mutations left are enough to overcome the initial parsing steps and then concentrate on common library code, which is also what we expect the probability distribution to converge to in later phases in the adaptive variant.

In all experiments, DARWIN outperformed AFL-S also after 200 minutes. In the six experiments where AFL-S eventually reached the same (average) coverage DARWIN reached after 200 minutes, it took AFL-S 285 more minutes on average. Further, in four experiments, AFL-S never even reached that mark.

B. Parameter Selection

Even though the parameters for μ and λ are widely consistent throughout literature [6], [31], we also evaluated neighboring configurations, as shown in Table IV. Our 24h experiments over ten runs show that within the large body of coverage evaluation targets, the initial configuration still outperforms them.

Orthogonality to Advanced Fuzzing Methods. To highlight DARWIN's benefit in more recent fuzzers, we extend EcoFuzz [67] with our mutation scheduler. EcoFuzz optimizes AFL's power schedule process to reduce AFL's focus on high-frequency paths. Within the fuzzer, we added four invocations to the DARWIN interface at the appropriate places in the code. We conducted a FuzzBench coverage experiment with 10 runs, 6h each. The full results are depicted in Table V. EcoFuzz-DARWIN outperforms its baseline in all but four experiments. While `libjpeg-turbo-07-2017` and `systemd_fuzz-link-parser` are quite close, the other two experiments show a larger difference. The DARWIN variant cannot outperform its baseline in `libxml2-v2.9.2` and `openthread-2019-12-23 openssl_x509`, similar as in the previous coverage experiment. Based on our investigation, this is also caused by the strongly structured input (openthread is an implementation of the OpenThread networking protocol), where the baseline fuzzer profits from higher execution speeds.

C. Execution Speed versus Efficiency

Challenge C.4 underlines the difficulty of optimizing probability distribution without spending too much time on a learning algorithm. This is important as an optimal distribution does not lead to a measurable improvement if the optimal selection can be found via brute force in less time. As such, we measure the effectiveness of the mutation scheduler in finding a good mutation probability distribution. Further, we analyze and compare the execution speed of DARWIN's ES, MOPT's PSO, and AFL's random sampling with a uniform probability distribution.

Scheduling Effectiveness. While it is rather simple to measure the effects of an algorithmic change in fuzzing via coverage or crash analysis, the resulting numbers are hard to attribute to the algorithmic change itself due to the fuzzers complexity. Hence, we derived a metric to directly capture the impact of mutation scheduling, namely the average number of mutations needed to go from one coverage point to another. Here, we get 1981.90 mutations for AFL, 1484.81 for MOPT, and 1491.32 for DARWIN. This clearly shows the advantage of mutation scheduling. MOPT and DARWIN achieve very similar results, where we attribute the difference to noise. The remaining question is whether both fuzzers also achieve the same execution speed, as the mutation schedulers' efficiency depends on both factors.

Performance Measurements. Table VI presents the observed execution speed over ten runs. Notably, AFL has the most executions, which makes sense considering that both DARWIN and MOPT add an optimization algorithm on top of AFL's random sampling; yet, the DARWIN's execution speed is relatively close to random selection. However, the numbers demonstrate that DARWIN is 48.26% (geometric mean) faster than MOPT while outperforming both other fuzzers in terms of coverage. This makes DARWIN solve Challenge C.4 and also highlights that the representation encoding for ES does not induce a major performance overhead.

This underlines that (1) DARWIN's mutation scheduler improves efficiency compared to uniform random sampling and (2) that DARWIN's mutation scheduler achieves this with less computational overhead than MOPT, addressing Challenge C.2 In conclusion, DARWIN has the same scheduling effectiveness but is much faster than MOPT, resulting in better efficiency.

D. MAGMA - Time-to-Bug Evaluation

MAGMA [32] is a recently published fuzzer benchmark that emphasizes the capability to uncover bugs, in particular, the time needed to reach a bug within a target. For this, the authors forward-port real-world bugs into current versions of tools used in practice, namely `libpng`, `libtiff`, `libxml2`, `openssl` (which we could not get to run with the current version of MAGMA on GitHub at the time of writing, `php`, `poppler`, and `sqlite3`. Further, MAGMA provides a framework around these tools to detect when a fuzzer reaches and triggers such a forward-ported bug. Hence, MAGMA's attempt to measure the time to reach a bug gives a much clearer picture of a fuzzer's efficiency in practice, as code coverage is merely a proxy metric to measure a fuzzer's success. We set up five hours fuzzing campaigns for each target for the MAGMA benchmark and repeated each experiment three times.

The results are depicted in Figure 7. Out of 21 bugs found in total, DARWIN can find 15 of them the fastest. MOPT is in 4 cases the fastest, but only because in two of them DARWIN could not trigger the bug (where MOPT is expected to take more than two days to find the bug on average). Finally, AFL can only find 12 bugs, further emphasizing that DARWIN increases the efficiency of the mutation selection.

E. Crashes

This final experiment explores DARWIN's ability to find crashes in well-fuzzed targets, which is commonly done to evaluate fuzzers [67], [39], [58], [52]. Note that our experiment differs from the setup MOPT paper to increase statistical meaningfulness. In the MOPT paper, the authors use 100 seed files per target. This, however, makes interpretation of the resulting data highly challenging, as the outcome heavily depends on which seed has been scheduled (also makes finding

TABLE IV. COVERAGE RESULTS MEASURED IN UNIQUE PATHS AND EDGES FOR WELL-FUZZED TARGETS IN BINUTILS OVER 10 RUNS, 24H EACH.

	DARWIN (μ:5 λ:4)		μ:5 λ:3		μ:5 λ:5		μ:6 λ:4		μ:4 λ:4	
Benchmark	unique paths	edges	unique paths	edges	unique paths	edges	unique paths	edges	unique paths	edges
`cxxfilt`	3334.18	2327.27	**3375.10**	**2365.70**	3233.50	2301.00	3251.70	2323.00	3224.80	2294.10
`objcopy`	**5760.82**	**7912.36**	5567.10	7866.10	5564.70	7835.60	5584.20	7821.60	5565.10	7813.90
`objdump`	**6018.91**	**7269.82**	5832.80	7239.20	5820.00	7256.00	5774.30	7221.60	5880.90	7244.40
`readelf`	29715.64	**13012.36**	**29821.00**	12990.70	29101.90	12813.30	29409.20	12928.80	29551.10	12934.20
`size`	**3020.91**	**4030.91**	2984.20	3979.70	2999.60	3990.60	2975.20	4022.80	2925.60	4018.80
`strip`	**5732.55**	7703.55	5533.70	7667.60	5608.8	**7716.20**	5542.80	7693.30	5549.30	7696.90

TABLE V. MEDIAN RELATIVE CODE COVERAGE ON EACH BENCHMARK AFTER 10 RUNS WITH 6H EACH. MEDIAN RELATIVE PERFORMANCE OF EACH FUZZER TO THE ENCOUNTERED EXPERIMENT MAXIMUM.

	EcoFuzz-DARWIN	EcoFuzz
FuzzerMedian	**97.31**	95.43
FuzzerMean	**94.47**	94.29
bloaty_fuzz_target	**95.86**	91.39
curl_curl_fuzzer_http	**97.31**	95.89
freetype2-2017	**95.92**	95.79
harfbuzz-1.3.2	**97.36**	94.12
libjpeg-turbo-07-2017	84.67	**86.47**
libpng-1.2.56	**97.83**	96.51
libxml2-v2.9.2	78.84	**93.66**
libxslt_xpath	**94.03**	93.97
mbedtls_fuzz_dtlsclient	**99.23**	96.71
openssl_x509	**99.67**	99.62
openthread-2019-12-23	80.11	**98.45**
php_php-fuzz-parser	**99.62**	99.49
proj4-2017-08-14	**90.53**	84.77
re2-2014-12-09	**98.83**	98.50
sqlite3_ossfuzz	**95.78**	83.01
systemd_fuzz-link-parser	97.96	**98.90**
vorbis-2017-12-11	**96.09**	95.43
woff2-2016-05-06	**97.60**	93.60
zlib_zlib_uncompress_fuzzer	**97.70**	95.24

TABLE VI. AVERAGED EXECUTIONS PER SECOND REACHED WITH THE RESPECTIVE MUTATION SCHEDULING APPROACH.

Benchmark	havoc	afl	mopt
bsdtar	2631.86	2385	1185.87
cxxfilt	2060.07	3766.8	2888.56
djpeg	2830.72	2609.9	1097.79
jhead	5097.98	5484.94	1679.58
objcopy	2019.37	2086.37	1867.22
objdump	1908.94	1932.47	1887.44
readelf	2439.79	2715.96	2389.04
size	2082.11	2147.08	1945.45
strip	2005.56	2159.54	1876.9
tcpdump	5042.22	5232.37	1554.85
geomean		-7.6%	+48.26%

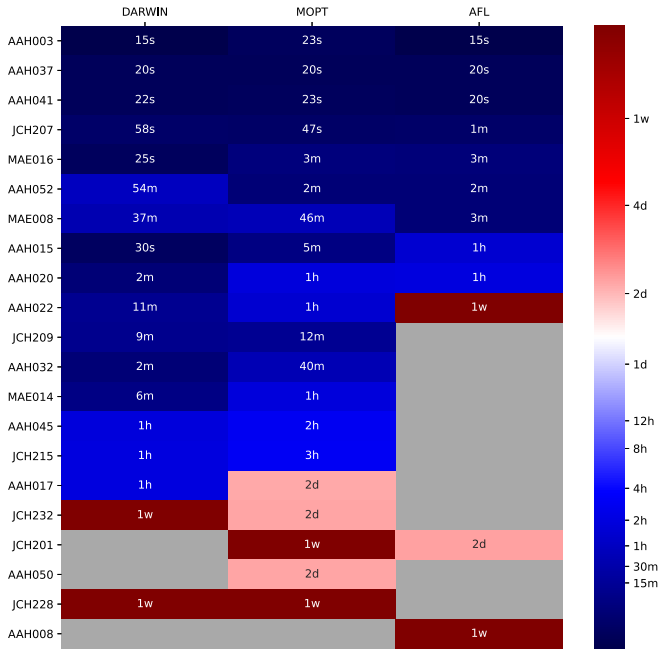


Fig. 7. The expected time to reach a bug in the MAGMA benchmark over three runs. Y axis shows the individual bugs. Lower time is better, grey indicates that a fuzzer has not found this bug.

novel bugs much more likely). Additionally, the experiment was running only once. Here, We conducted a 24h experiment with 10 runs (and same seeds as in previous experiments) to also evaluate the stability in finding bugs. We use the same benchmarks as used in Section VI-A already.

The resulting crashes are shown in Table VII. Then, we minimized the test cases using afl-tmin and verified them with afl-collect [53]. Then we first removed test cases with the same MD5 hash, and the Address Sanitizer output refers to the same line. Finally, we manually verified that they differ and lead to a crash, which we refer to as "triaged" in Table VII. In total, we found 20 unique bugs with DARWIN, and 26 unique bugs with the DARWIN-enhanced version of EcoFuzz. In contrast, the baselines, AFL and EcoFuzz, only found 12 resp. 1 unique bug(s). Also, the stability of their findings (i.e., the mean over all runs) is way below the DARWIN-based fuzzers.

DARWIN also found a completely novel bug in `objcopy` (working up to binutils 2.39, introduced more than 24 years ago), which is leading to a memory leak. `copy_relocations_in_section` in objcopy.c is not freeing a buffer (relpp) in every possible case. This bug is very hard to trigger, as the function is only called at high stack depths. The testcase leading to the bug was found through splicing based on a relatively early testcase and a testcase from the middle of the experiment. We responsibly disclosed the triaged bug to the respective developers, who acknowledged and fixed the bug [3].

VII. RELATED WORK

Fuzzing is an active research domain but is also widely used in practice. It has been improved in various areas, e.g., grammar-based fuzzing that also might use mutations [3], dedicated mutations, or program transformations for common roadblocks [4], [63], [49], or fuzzers for hard-to-fuzz software, e.g., due to hardware dependencies [69], [20]. We consider these works orthogonal to ours. Next, we restrict ourselves to the areas of mutation scheduling but also seed-selection

[3]https://sourceware.org/bugzilla/show_bug.cgi?id=29233

TABLE VII. CRASHES ENCOUNTERED IN A 24H CAMPAIGN OVER 10 RUNS. ALL BUGS ARE TRIAGED CRASHES. "MAX" REFERS TO THE MAXIMUM ENCOUNTERED BUGS WITHIN A RUN. "UNIQ" REFERS TO THE NUMBER OF UNIQUE CRASHES OVER ALL 10 RUNS.

	DARWIN			AFL			AFL-S			MOPT			EcoFuzz-D			EcoFuzz		
Benchmark	mean	max	uniq	mean	max	uniq	mean	max	uniq	mean	max	uniq	mean	max	uniq	mean	max	uniq
bsdtar	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
djpeg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
tcpdump	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
jhead	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
readelf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
strip	0.25	**3**	**3**	0.09	1	1	**0.3**	2	**3**	0	0	0	0	0	0	0	0	0
size	**0.92**	**2**	**11**	0.45	1	5	0.7	1	7	0.17	1	2	0.3	1	3	0.1	1	1
filt	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
objdump	**0.25**	**1**	**3**	0.09	1	3	0.1	1	1	0	0	0	0.2	1	2	0	0	0
objcopy	**0.25**	1	3	0.18	1	3	0.1	1	1	0	0	0	**2.1**	**17**	**21**	0	0	0
Total		**7**	**20**		4	12		5	12		1	2		**18**	**26**		1	1

algorithms, as the underlying approaches are often similar. Finally, we compare DARWIN to the presented works.

A. Mutation Strategies

Mutation strategies try to optimize either what mutations should be applied (which we refer to as *mutation scheduling*) or where in the input those mutations should be applied (which we refer to as *location optimization*).

Mutation Scheduling. In 2018, two works proposed to leverage machine learning approaches to improve mutation scheduling. Böttinger et al. used deep Q-learning (a type of reinforcement learning) to find policies that can next generate new higher reward inputs [10]. Drozd and Wagner optimized mutation operators using reinforcement learning to achieve deeper coverage across several varied benchmarks [15]. Despite leveraging complex algorithms, both of those works do not manage to show significant improvements in vulnerability discovery, underlining that the algorithms are too complex to address Challenges C.1 and C.4. Lyu et al. considered a different approach for optimizing mutation scheduling and proposed a mutation scheduling scheme called MOPT [39]. MOPT was the first work to propose using heuristic techniques for optimizing general mutation scheduling. More precisely, MOPT uses a custom variant of Particle Swarm Optimization (PSO) to approximate the best selection probability distribution for mutation operators. We note that for PSO, there is no guarantee to converge to the global optimum (only to the best particle in the swarm) [61], [18]. At the same time, there are proofs of convergence for evolution strategy [31]. Further, MOPT proposes to deactivate the deterministic fuzzing stage either temporarily or permanently to make PSO converge faster.

As this work is closest in the objective and applied techniques to ours, we discuss the main differences between MOPT and DARWIN in more detail . From MOPT's design perspective, the authors do not show how several parameters need to be tuned to reach a good performance under which condition. In particular, it is not evaluated how many solutions (swarms) are needed in practice and how difficult it is to tune them, or how sensitive those parameters are. Hence, MOPT does not solve Challenge C.3. Since the MOPT algorithm has both local and global positions for particles, the algorithm requires additional measures to find the best solutions, increasing the complexity of the algorithm. This leads to a performance reduction in the havoc stage, as we explore in Section VI-C. Thus, MOPT cannot address Challenge C.4. A change of solution encoding, as proposed in Section IV-C, requires changes in MOPT 's algorithm. Finally, what the authors call a swarm is actually a solution in a swarm. What the authors denote as multiple swarms is one swarm.

In contrast, DARWIN has no parameters to tune from the fuzzer side. ES has only two parameters, μ and λ, which are intuitive to select during fuzzer development time and have a clear role in the evolution process. DARWIN does not require any additional communication between modules to run the evolution process. DARWIN uses a simple fitness function where the goal is the maximization of the code coverage. DARWIN supports various solution representations without requiring changes in the DARWIN algorithm. We develop DARWIN not only to be well-performing for the specific application at hand but also to conform to standards from the EA community regarding the design choices and performance evaluation.

From the performance perspective, MOPT's PSO integration is computationally intense (i.e., already reducing coverage significantly over time due to decreased speed), and the evaluation does not explore whether a simpler algorithm or even a static distribution might already be enough. The evaluation results are also produced by a varying amount of seed files, but not a typical setup with one empty and one small seed suitable for the application. Further, MOPT's mutation scheduling algorithm is not evaluated separately from the other stages of the fuzzer but always with the deterministic stage running at least once.

Our evaluation shows that these two aspects distorted the comparison with the default random mutation selection by microbenchmarking the mutation selection using our proposed average-mutations to a new coverage metric. Besides, the huge size of seeds might lead to a distortion in coverage measurements since a fuzzer might be stuck for a while given a bad randomly chosen seed. While we consider MOPT's pacemaker mode as orthogonal, we still show that with a permanently disabled deterministic stage, AFL discovers significantly more unique paths than MOPT, which is in line with the results reported in Google's FuzzBench [27]. In contrast, DARWIN's selection algorithm is much simpler, has, thanks to its more lightweight Evolution Strategy and solution representation, a lower impact on execution speed, brings a measurable improvement over the standard uniform mutation selection, and even outperforms MOPT significantly in terms of coverage and crashes found.

Location Optimization. In contrast to mutation scheduling approaches, some works aim to find the right locations in the inputs to mutate. One example is FairFuzz which applies a deterministic combination of mutations to explore which bytes in the test case reach rare branches when mutated [37]. These bytes now form a mask used in the havoc stage to (partially) limit mutation operators to these bytes. A similar approach has been proposed by Rajpal et al. [51], where neural networks are used to infer (un-)promising bytes in inputs generated by past mutations. Promising bytes are then preferred during mutation. Another work, Steelix [38], leverages static analysis to extract information about comparisons in the target program, which is then used to mutate responsible bytes in the input efficiently. Analogous to FairFuzz, the information generated by the static analysis is used to create a mask. If a mutated input does not generate new coverage, but a byte in the mask is closer to what the comparison expects, this byte is further mutated. All of these approaches above focus on where to apply mutations, whereas DARWIN optimizes general mutation selection. Further, many of the mentioned ideas can be combined with our approach.

B. Seed-selection Algorithms

Seed-selection algorithms aim to distill and select a subset of seeds to optimize for a specific branch to pass or improve coverage in general by preferring more promising seeds or minimizing seeds to improve execution speed. MoonShine uses system call traces of real-world programs to distill them into a minimal test case that still achieves 86% of the pre-distilled coverage [48]. These minimal tests can then be used to 1) trigger basic blocks that require a certain order of system calls and 2) improve the fuzzing speed.

A similar idea is used by FasterFuzzing, which employs a Generative Adversarial Network trained with an initial seed corpus to generate new, better seeds [44]. EcoFuzz [67] proposes a seed scheduling algorithm to fine-tune exploration and exploitation. After a short fuzzing period, EcoFuzz switches to the exploration phase, where the remaining seeds are fuzzed to estimate their reward probability. Then, EcoFuzz switches to the exploitation phase to fuzz these seeds that have the highest reward probability. If a new path has been discovered, EcoFuzz switches back to the exploration phase. This increases coverage while reducing the number of test case generations. AFLFast identifies that fuzzers are often stuck with high-frequency paths [8]. To balance this, AFLFast leverages a Markov model to identify and prefer low-frequency paths as a heuristic. Similarly, VUzzer uses an evolutionary algorithm approach to leverage control-flow features and find hard-to-reach paths while also avoiding inputs that reach basic blocks containing error-handling code [52]. NeuFuzz, instead, does not try to balance low- and high-frequency paths but uses a neural network to prefer paths that are prone to contain vulnerabilities [64]. Angora follows a more general strategy by preferring inputs that lead to unexplored branches, effectively also balancing high- and low-frequency path exploration [12]. AFLSmart uses a structural representation of seed to perform semantically correct mutations and increases time spent on mutating promising seeds that pass the input parsing [50]. AFLGo [7] enables directed fuzzing close to chosen target locations by prioritizing seeds that reach paths close to the target [7]. Seed-scheduling and -distilling algorithms optimize an early stage in the fuzzing process. Hence, it is challenging for these techniques to steer the mutation phase unless the havoc stage is specifically aware of, e.g., the phases defined in EcoFuzz. This might lead to counterproductive mutations being applied to optimized seeds, canceling out the desired effect. In contrast, DARWIN optimizes a late stage in the fuzzing process and thus, can learn a favorable probability distribution to keep the properties of promising inputs.

C. Algorithmic Improvements vs. Optimizing Execution Speed

Many works recently focused on the raw speed of input generation and mutation with big coverage improvements [56], [19], [4], [30]. While DARWIN offers fewer coverage improvements as reported by these fuzzers, DARWIN's mutation scheduling is orthogonal to performance increases achieved through, e.g., fast snapshotting. Hence, DARWIN can further increase coverage, and more importantly—as we show in Appendix D and Section VI-D—improve the bug triggering capabilities of these fuzzers.

VIII. Conclusion

We presented DARWIN, a novel mutation scheduling algorithm that uses an Evolution Strategy to optimize the mutation selection probability distribution based on the instrumented application's feedback. DARWIN tackles all of our identified challenges in building a mutation scheduler: Challenge C.1 by integrating Evolutionary Strategy as a mutation scheduler, significantly outperforming the state-of-the-art mutation scheduler MOPT [39], while also being the first mutation scheduler to show a significant increase in edge coverage of 1.73% over AFL respectively, bugs uncovered in both, LAVA-M and MAGMA, and decrease in time to find bugs over AFL and MOPT; Challenge C.2 by choosing reasonable encoding and parameters; Challenge C.3 by introducing no user-facing parameters that need to be tuned per target; and Challenge C.4 by maintaining a high execution speed compared to the AFL baseline, in contrast to related work, which is far slower. Further, DARWIN found 20 unique bugs in widely-used real-world applications, outperforming both AFL and MOPT. DARWIN was the only fuzzer able to also uncover a new bug that is still working on the most recent version of the target. While our experiments show that unique path coverage for fitness provides good feedback for ES, other heuristics could be used. For example, it would be interesting to include the number of crashes and consider the Pareto fronts of the solutions. Further, future research could study the efficiency of multi-objective algorithms for mutation scheduling that combine several of the previous suggestions, e.g., also include the frequency of the path to focus more on low-frequency paths or the block hit count to promote stronger intensification.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research in the StartUpSecure funding program "Sanctuary" (16KIS1417), the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research and the Arts within ATHENE, and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 952697).

REFERENCES

[1] Mohamed Abouhawwash, Kalyanmoy Deb, and Adam Alessio. Exploration of multi-objective optimization with genetic algorithms for pet image reconstruction. *Journal of Nuclear Medicine*, 61(supplement 1):572–572, 2020.

[2] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[5] H.-G. Beyer and B. Sendhoff. Evolution strategies for robust optimization. In *2006 IEEE International Conference on Evolutionary Computation*, pages 1346–1353, 2006.

[6] Hans-Georg Beyer and Hans-Paul Schwefel. *Evolution Strategies – A Comprehensive Introduction*, volume 1. Kluwer Academic Publishers, USA, May 2002.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[9] Jürgen Branke, Su Nguyen, Christoph W. Pickardt, and Mengjie Zhang. Automated design of production scheduling heuristics: A review. *IEEE Transactions on Evolutionary Computation*, 20(1):110–124, 2016.

[10] K. Böttinger, P. Godefroid, and R. Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 116–122, 2018.

[11] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108, 2018.

[12] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*, 2018.

[13] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 633–645, 2019.

[14] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[15] William Drozd and Michael D. Wagner. Fuzzergym: A competitive framework for fuzzing and learning. *CoRR*, abs/1807.07490, 2018.

[16] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer-Verlag, Berlin Heidelberg New York, USA, 2003.

[17] Michael Emmerich, Ofer M Shir, and Hao Wang. *Evolution Strategies*, chapter 4, pages 1–31. Springer International Publishing, 2018.

[18] Andries P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence*. Wiley, 2005.

[19] Brandon Falk. Vectorized emulation: Hardware accelerated taint tracking at 2 trillion instructions per second. https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html. Accessed: 2022-04-26.

[20] Bo Feng, Alejandro Mera, and Long Lu. P 2 im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[21] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[22] Abhinav Gaur, A.K.M. Khaled Talukder, Kalyanmoy Deb, Santosh Tiwari, Simon Xu, and Don Jones. Unconventional optimization for achieving well-informed design solutions for the automobile industry. *Engineering Optimization*, 52(9):1542–1560, 2020.

[23] Morteza Gholamipoor, Parviz Ghadimi, Mohammad H. Alavidoost, and Mohammad A. Feizi Chekab. Application of evolution strategy algorithm for optimization of a single-layer sound absorber. *Cogent Engineering*, 1(1):945820, 2014.

[24] Abhiroop Ghosh, Erik Goodman, Kalyanmoy Deb, Ronald Averill, and Alejandro Diaz. A large-scale bi-objective optimization of solid rocket motors using innovization. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2020.

[25] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, USA, 1997.

[26] Fred W. Glover and Gary A. Kochenberger, editors. *Handbook of Metaheuristics*, volume 114 of *International Series in Operations Research & Management Science*. Springer, 1 edition, January 2003.

[27] Google. Fuzzbench: 2020-09-28 report. https://www.fuzzbench.com/reports/2022-04-19/index.html. Accessed: 2022-04-26.

[28] Google. Oss-fuzz. https://google.github.io/oss-fuzz/. Accessed: 2022-04-26.

[29] Google. *american fuzzy loop (afl)*. https://github.com/google/AFL, 2020.

[30] Rahul Gopinath and Andreas Zeller. Building fast fuzzers. *arXiv preprint arXiv:1911.07707*, 2019.

[31] Nikolaus Hansen, Dirk V. Arnold, and Anne Auger. *Evolution Strategies*, pages 871–898. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[32] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[33] Xiaolin Hu, Carlos A Coello Coello, and Zhangcan Huang. A new multi-objective evolutionary algorithm: Neighbourhood exploring evolution strategy. *Engineering Optimization*, 37(4):351–379, 2005.

[34] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[35] Walter Krawec, Stjepan Picek, and Domagoj Jakobovic. Evolutionary algorithms for the design of quantum protocols. In Paul Kaufmann and Pedro A. Castillo, editors, *Applications of Evolutionary Computation*, pages 220–236, Cham, 2019. Springer International Publishing.

[36] P. J. M. Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers, USA, 1987.

[37] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[38] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.

[39] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[40] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

[41] Microsoft. Microsoft announces new project onefuzz framework, an open source developer tool to find and fix bugs at scale. https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/. Accessed: 2022-04-26.

[42] Julian F. Miller. *Cartesian Genetic Programming*, pages 17–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[43] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[44] Nicole Nichols, Mark Raugas, Robert Jasper, and Nathan Hilliard. Faster fuzzing: Reinitialization with deep neural models. *arXiv preprint arXiv:1711.02807*, 2017.

[45] Beatrice Ombuki-Berman and Franklin Hanshar. *Using Genetic Algorithms for Multi-depot Vehicle Routing*, volume 161, pages 77–99. Springer Berlin Heidelberg, 09 2008.

[46] Mark A Overton. Romu: Fast nonlinear pseudo-random number generators providing high quality. *arXiv preprint arXiv:2002.11331*, 2020.

[47] Inc. OWASP Foundation. Owasp top ten 2017. https://owasp.org/www-project-top-ten/2017/A9_2017-Using_Components_with_Known_Vulnerabilities. Accessed: 2022-04-26.

[48] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[49] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[50] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[51] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[52] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[53] rc0r. afl-utils. https://gitlab.com/rc0r/afl-utils/-/tree/master/afl_utils. Accessed: 2022-04-26.

[54] Lino Rodriguez-Coayahuitl, Alicia Morales-Reyes, Hugo Jair Escalante, and Carlos A. Coello Coello. Cooperative co-evolutionary genetic programming for high dimensional problems. In Thomas Bäck, Mike Preuss, André Deutz, Hao Wang, Carola Doerr, Michael Emmerich, and Heike Trautmann, editors, *Parallel Problem Solving from Nature – PPSN XVI*, pages 48–62, Cham, 2020. Springer International Publishing.

[55] Claude Sammut and Geoffrey I. Webb, editors. *Particle Swarm Optimization*. Springer US, Boston, MA, 2010.

[56] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[57] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157. IEEE, 2016.

[58] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[59] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, 2019.

[60] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[61] Frans Van Den Bergh and A. P. Engelbrecht. *An Analysis of Particle Swarm Optimizers*. PhD thesis, ZAF, 2002. AAI0804353.

[62] Dmitry Vyukov. syzkaller - kernel fuzzer. https://github.com/google/syzkaller. Accessed: 2022-04-26.

[63] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[64] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, 7:36340–36352, 2019.

[65] Lichao Wu, Gerard Ribera, Noemie Beringuier-Boher, and Stjepan Picek. A fast characterization method for semi-invasive fault injection attacks. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, pages 146–170, Cham, 2020. Springer International Publishing.

[66] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *Proceedings of the International Conference on Software Engineering*, 2022.

[67] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[68] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[69] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[70] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2255–2269. USENIX Association, August 2020.

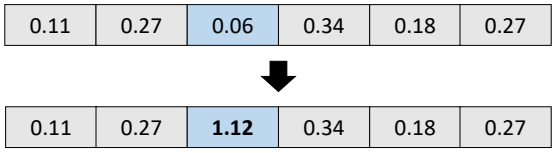
APPENDIX

A. Evolutionary Algorithms

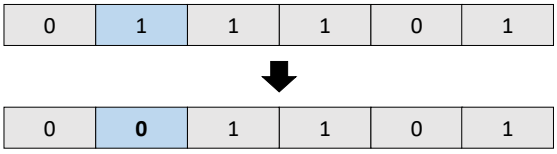
The pseudocode for evolutionary algorithms is given in Algorithm 3, while in Figures 8(a) and 8(b), we present mutations working on floating-point and binary encoding, respectively.

Algorithm 3 Pseudocode for EA.

$t \leftarrow 0$
$P(0) \leftarrow CreateInitialPopulation$
repeat
  $t \leftarrow t + 1$
  $P'(t) \leftarrow SelectionMechanism\ (P(t-1))$
  $P(t) \leftarrow VariationOperators(P'(t))$
until *TerminationCriterion*
$Return\ OptimalSolutionSet(P)$



(a) Depiction of perturbation for real-valued vector. The sum of all values does not need to be equal to 1 and every gene must have a non-negative value.



(b) Depiction of perturbation for binary vector.

Fig. 8. Perturbation operators for various solution encodings. The gene depicted in the blue color is mutated.

B. Experiments on Encoding & RNG

We depict the evaluation results for different encodings and RNGs in Table VIII

TABLE VIII. AVERAGED EXECUTIONS PER SECOND REACHED WITH THE RESPECTIVE VARIATION OF DARWIN. POSITIVE PERCENTAGES THAT DARWIN WAS THIS MUCH FASTER THAN THE FUZZER IN THE COLUMN.

	DARWIN	D-Std. RNG	D-Real Valued
Benchmark	execs/s	execs/s	execs/s
`cxxfilt`	2210.41	2151.47	1860.02
`objcopy`	2610.73	2630.80	2678.77
`objdump`	1687.52	2161.75	2225.95
`readelf`	3405.55	2711.17	2815.48
`size`	3140.08	2733.33	2910.92
`strip`	2686.19	2492.25	2665.44
geomean		+3.62 %	+2.42%

C. Seed Used for Binutils

We build a minimal ELF seed testcase for binutils to achieve adequate execution speed. Its code is depicted in Listing 1.

```
extern "C" void _start() {
    __asm("mov $60, %rax\n\t
           xor %rdi, %rdi\n\t
           syscall");
}
```

Listing 1. Source code for binutils seed, calling `sys_exit`.

D. LAVA-M - Finding Known Bugs

LAVA-M [14] is a synthetic set of bugs inserted into the GNU coreutils suite. These hard-to-reach bugs are injected automatically into the real-world binaries `who`, `uniq`, `md5sum`, and `base64`. While LAVA-M has questionable implications on real-world performance, it is commonly used to evaluate fuzzers in research [4], [52], [49], [12], [64], [39]. As LAVA-M is heavily focusing on comparisons, LAVA-M favors approaches that concentrate on improving mutation operators themselves [4]. Hence, we keep this for the sake of completeness here in the abstract. While the benchmark provides one initial test case per target, we added an uninformed, empty test case for each target to be consistent with our other experiments. Each target is fuzzed for five hours, as commonly done for the LAVA-M benchmark in fuzzing papers [39], [4], [52]. Table IX depicts the results for DARWIN, MOPT, and AFL over three runs. Notably, DARWIN is the only fuzzer in our evaluation that finds bugs across all targets and consistently finds the highest number of bugs in each target. For `uniq` and `who`, which are the only targets where all fuzzers found bugs, we further analyze in which fuzzing loop stage the bugs were found. In the case of `uniq`, DARWIN finds 50% of the bugs using the havoc stage, while MOPT and AFL exclusively found all bugs using splicing. For `who`, the havoc stage attributes for one-third of the bugs found by DARWIN, whereas on AFL, the havoc stage accounts for 20% of the bugs. On MOPT, the havoc stage is never successful in finding a bug, possibly because some mutators are never scheduled. By comparing the maximum numbers found per fuzzer, we can conclude that DARWIN found more bugs than just the overlap between all fuzzers. Finally, DARWIN's approach for mutation scheduling is orthogonal to, e.g., improvements in overcoming branch checks [12], [68], [4], and can be used to optimize the scheduling of the respective mutation operators to achieve a synergetic effect.

TABLE IX. CRASHES FOUND IN LAVA-M, AVERAGE CRASHES OVER THREE RUNS AS WELL AS THE HIGHEST NUMBER OF CRASHES ENCOUNTERED WITHIN AN INDIVIDUAL RUN.

	DARWIN		MOPT		AFL	
Benchmark	Avg.	Max.	Avg.	Max.	Avg.	Max.
`base64`	1	2	0	0	0.33	1
`md5sum`	0.33	1	0.33	0	0	0
`uniq`	3.67	4	0.33	1	0.33	1
`who`	3	3	2	2	2.67	3
Total	8	10	2.67	3	2.33	1

E. Mutations in the AFL Havoc Stage

Table X lists all mutations defined in the AFL havoc stage.

ID	Description
0	Flip single bit
1	Set byte to interesting value
2	Set word to interesting value
3	Set dword to interesting value
4	Randomly subtract from byte
5	Randomly add to byte
6	Randomly subtract from word
7	Randomly add to word
8	Randomly subtract from dword
9	Randomly add to dword
10	Set a random byte to a random value
11	Delete Bytes
12	Delete Bytes
13	Clone bytes (75%) or insert a block of constant bytes (25%)
14	Overwrite bytes with a randomly selected chunk (75%) or fixed bytes (25%)
15	Overwrite bytes with an extra
16	Insert an extra

TABLE X. MUTATIONS DEFINED IN THE AFL HAVOC STAGE, DESCRIPTIONS TAKEN FROM THE AFL SOURCE CODE [29]. EXTRA REFERS TO TARGET-SPECIFIC DICTIONARY ENTRIES. 11 AND 12 TRIGGER THE SAME MUTATION TO INCREASE SELECTION PROBABILITY BASED ON PRACTICAL EXPERIENCE.