System Architecture Designs for Secure, Flexible and Openly-Accessible Enclave Computing

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

MSc. Emmanuel Simon Stapf

geboren in Heidelberg, Deutschland

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

Tag der Einreichung: 28.2.2022
Tag der Disputation: 24.5.2022

TECHNISCHE
UNIVERSITÄT
DARMSTADT

System Security Lab
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

Darmstadt 2022

Emmanuel Simon Stapf:
System Architecture Designs for Secure, Flexible and Openly-Accessible Enclave Computing
© February 2022

Darmstadt, Technical University of Darmstadt
Day of dissertation defense: 24.05.2022
Dissertation publication at TUprints: 2022

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. Reiner Hähnle
Dr. Jean-Paul Degabriele

Erklärung gemäß §9 der Promotionsordnung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendung 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, Februar 2022

Emmanuel Simon Stapf

Abstract

In the last decade, security architectures became prominent which protect sensitive data in isolated execution environments, called enclaves or Trusted Execution Environments (TEEs), that are backed by hardware-assisted security mechanisms. Relying on hardware mechanisms allows enclave architectures to shrink the software that is inherently trusted, called Trusted Computing Base (TCB), to a bare minimum which stands in stark contrast to the large code base that must be trusted in a commodity operating system. Moreover, in contrast to architectures which deploy security hardware in dedicated computer chips, e.g., Trusted Platform Modules (TPMs) or smart cards, enclave architectures are deeply integrated into the main processor and thus can utilize the full computational power of the processor while still reducing hardware costs. Even though enclave architectures are widely deployed in computing systems, ranging from resource-constraint microcontrollers and embedded systems over mobile devices to personal computers and servers, still many challenges must be solved to enable their full potential.

In this dissertation, we design, implement and evaluate multiple novel enclave architectures and security extensions which contribute significantly to enclave computing research by tackling multiple research challenges, namely i) providing an open access to enclave computing on ARM-based systems, ii) protecting diverse sensitive applications with a single enclave architecture across platforms, and iii) providing side-channel resilient enclaves.

Openly-accessible Enclave Computing on ARM-based Devices. ARM TrustZone was one of the first security technologies which enabled enclave computing. Its wide deployment on mobile devices has the potential to guarantee security for many sensitive mobile applications. Unfortunately, the current enclave architectures based on TrustZone cannot provide enclave protection for all sensitive applications since each protected application increases the attack surface of the system. As a result, enclave computing on ARM-based devices is today largely blocked for third-party application developers and mostly used for services from the device vendors. In our work we propose SANCTUARY, a novel enclave architecture design which enables enclave protection for all sensitive applications. SANCTUARY achieves this by providing de-privileged enclaves based on a strong hardware-assisted isolation without requiring to modify hardware. SANCTUARY's enclaves do not increase the attack surface of the system and thus solve the challenge of making TrustZone available to all application developers. We implement and evaluate a prototype of SANCTUARY on an off-the-shelf ARM-based multi-core chip set.

By making TrustZone openly accessible, SANCTUARY can provide protection for a multitude of applications which process privacy-sensitive data. In Offline Model Guard

(OMG), we implement an offline keyword recognition service in a Sanctuary enclave which guarantees, privacy for the user's speech data, integrity for the machine learning algorithms and confidentiality for the machine learning models which represent an important intellectual property for the service provider.

This part of the dissertation is based on the following publications:

[22] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **SANCTUARY: ARMing TrustZone with User-space Enclaves**. In *Symposium on Network and Distributed System Security (NDSS)*, 2019. CORE Rank A*. Appendix A.

[12] 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 *Conference on Design, Automation and Test in Europe (DATE)*, pages 460-465, 2020. CORE Rank B. Appendix B.

Flexible Enclaves Across Platforms. All existing enclave architectures make assumptions regarding the applications they envision to protect. So far, this has led to an *one-size-fits-all* design paradigm in which each enclave architecture provides exactly one *type* of enclave with inherent capabilities and limitations. As a result, enclave architectures are either restricted in the applications they can protect or require workarounds to circumvent the weaknesses of a particular enclave type. In our work Cure, we forsake the one-size-fits-all paradigm and propose the first enclave architecture which provides multiple types of enclaves in order to fulfill the individual requirements of sensitive applications. Cure advances the state-of-the-art in enclave architectures and introduces new hardware security mechanisms which allow to assign system resources, e.g., processor cores, cache memory or peripherals flexibly to enclaves. We design and implement a prototype of Cure for the open RISC-V architecture and evaluate its hardware and performance overhead on an FPGA- and simulator-based evaluation setup.

On emerging heterogeneous computing platforms, potentially hundreds of diverse computing nodes are connected over a Network-on-Chip (NoC) architecture. Enabling enclave computing also on NoC-based architectures brings many new security challenges, e.g., regarding the inclusion of computing nodes from untrusted third-party vendors or the distribution of sensitive enclave data across multiple nodes. In our work, we combine Cure's multi-type enclave concept with a novel hardware security component, the Distributed Memory Guard (DMG), to design the first ever enclave architecture for NoC-based platforms which is also capable of dealing with untrusted computing nodes and distributed enclave data.

This part of the dissertation is based on the following publications:

[9] 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 *USENIX Security Symposium*, 2021. CORE Rank A*. Appendix C.

[61] Ghada Dessouky, Mihailo Isakov, Michel A. Kinsy, Pouya Mahmoody, Miguel Mark, Ahmad-Reza Sadeghi, Emmanuel Stapf, and Shaza Zeitouni. **Distributed Memory Guard: Enabling Secure Enclave Computing in NoC-based Architectures**. In *ACM Design Automation Conference (DAC)*, pages 985-990, 2021. CORE Rank A. Appendix D.

New Cache Designs enabling Side-Channel Resilient Enclaves. A large body of security research has shown the severity of cache side-channel attacks across computing systems, including enclave architectures. However, none of the deployed enclave architectures provide cache side-channel resilient enclaves which is a great deficiency. Cache architectures which allow to strictly partition and assign cache resources to execution contexts are a promising approach to provide strong side-channel security guarantees. Unfortunately, existing cache partitioning designs can either not prevent all attacks or separate the cache only into coarse-grained partitions which leads to an underutilization of the cache resources and makes the designs unable to scale to a large number of execution contexts. In our work CHUNKED-CACHE, we propose a novel cache microarchitecture designed for enclave architectures which allows to assign cache resources exclusively on-demand to enclaves on a fine-grained cache-set basis. Thus, CHUNKED-CACHE can support a large number of enclaves while providing the same cache utilization strategy used in commodity cache architectures. We implement a prototype of CHUNKED-CACHE in hardware and on a cycle-accurate cache simulator and show its small hardware and performance overhead during our evaluation.

This part of the dissertation is based on the following publication:

[62] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **CHUNKED-CACHE: On-Demand and Scalable Cache Isolation for Security Architectures**. In *Symposium on Network and Distributed System Security (NDSS)*, 2022. CORE Rank A*. Appendix E.

Zusammenfassung

In den letzten zehn Jahren haben sich Sicherheitsarchitekturen durchgesetzt, die sensible Daten in isolierten Ausführungsumgebungen, sogenannten Enklaven oder Trusted Execution Environments (TEEs), schützen, die auf hardwaregestützte Sicherheitsmechanismen aufbauen. Durch die Verwendung von Hardwaremechanismen können Enklaven-Architekturen die inhärent vertrauenswürdige Software, Trusted Computing Base (TCB) genannt, auf ein Minimum reduzieren, was in starkem Gegensatz zu der großen Codebasis steht, der in einem Standard-Betriebssystem vertraut werden muss. Im Gegensatz zu Architekturen, bei denen die Sicherheitshardware in speziellen Computerchips, z. B. Trusted Platform Modules (TPMs) oder Smartcards, untergebracht ist, sind Enklave-Architekturen tief in den Hauptprozessor integriert und können so die volle Rechenleistung des Prozessors nutzen und gleichzeitig die Hardwarekosten senken. Obwohl Enklave-Architekturen in Computersystemen weit verbreitet sind, von ressourcenbeschränkten Mikrocontrollern und eingebetteten Systemen über mobile Geräte bis hin zu Personal Computern und Servern, müssen noch viele Herausforderungen gemeistert werden, um ihr volles Potenzial auszuschöpfen.

In dieser Dissertation entwerfen, implementieren und evaluieren wir mehrere neuartige Enklave-Architekturen und Sicherheitserweiterungen, die einen wichtigen Beitrag zur Enklave-Computing-Forschung leisten, indem sie mehrere Herausforderungen der Forschung meistern, nämlich i) einen offenen Zugang zum Enklave-Computing auf ARM-basierten Systemen zu ermöglichen, ii) unterschiedliche sensible Anwendungen mit einer einzigen Enklave-Architektur plattformübergreifend zu schützen und iii) vor Seitenkanalangriffen geschützte Enklaven bereitzustellen.

Offen zugängliches Enklave-Computing auf ARM-basierten Geräten. ARM TrustZone war eine der ersten Sicherheitstechnologien, die Enklave-Computing ermöglichten. Ihr breiter Einsatz auf mobilen Geräten besitzt das Potenzial, die Sicherheit vieler sensibler mobiler Anwendungen zu ermöglichen. Leider können die derzeitigen auf TrustZone basierenden Enklave-Architekturen keinen Enklave-Schutz aller sensiblen Anwendungen bieten, da jede geschützte Anwendung die Angriffsfläche des Systems vergrößert. Infolgedessen ist Enklave-Computing auf ARM-basierten Geräten heute für Anwendungsentwickler von Drittanbietern weitestgehend blockiert und wird hauptsächlich für die Dienste der Gerätehersteller genutzt. In unserer Arbeit schlagen wir Sanctuary vor, eine neuartige Enklaven-Architektur, die einen Enklaven-Schutz aller sensiblen Anwendungen ermöglicht. Sanctuary erreicht dies, indem es de-privilegierte Enklaven auf der Basis einer starken hardwaregestützten Isolation bereitstellt, ohne jedoch eine Modifikation der Hardware vorauszusetzen. Sanctuarys Enklaven erhöhen die Angriffsfläche des Systems nicht und meistern somit die Herausforderung, Trust-

Zone allen Anwendungsentwicklern zugänglich zu machen. Wir implementieren und evaluieren einen Prototyp von Sanctuary auf einem handelsüblichen ARM-basierten mehrkernigen Chipsatz.

Aufgrund der erreichten offenen Zugänglichkeit von TrustZone kann Sanctuary eine Vielzahl von Anwendungen schützen, die datenschutzsensible Daten verarbeiten. In Offline Model Guard (OMG) implementieren wir einen offline Worterkennungsdienst in einer Sanctuary-Enklave, welcher den Schutz der Nutzersprachdaten, die Integrität der maschinellen Lernalgorithmen und die Vertraulichkeit der gelernten Modelle, welche ein wichtiges geistiges Eigentum des Dienstanbieters darstellen, garantiert.

Dieser Teil der Dissertation basiert auf den folgenden Veröffentlichungen:

[22] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, und Emmanuel Stapf. **SANCTUARY: ARMing TrustZone with User-space Enclaves**. In *Symposium on Network and Distributed System Security (NDSS)*, 2019. CORE Rank A*. Appendix A.

[12] Sebastian P. Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, und Christian Weinert. **Offline Model Guard: Secure and Private ML on Mobile Devices**. In *Conference on Design, Automation and Test in Europe (DATE)*, Seiten 460-465, 2020. CORE Rank B. Appendix B.

Flexible Plattformübergreifende Enklaven. Alle bestehenden Enklaven-Architekturen treffen Annahmen über die Anwendungen, welche sie schützen wollen. Bislang hat dies zu einem Paradigma geführt, bei dem jede Enklaven-Architektur genau einen Enklaven-Typ mit inhärenten Fähigkeiten und Einschränkungen bereitstellt. Infolgedessen sind Enklave-Architekturen entweder in den Anwendungen, die sie schützen können eingeschränkt oder sie sind abhängig von Behelfslösungen um die Schwächen eines bestimmten Enklave-Typs zu umgehen. In unserer Arbeit Cure geben wir dieses Einheitsparadigma auf und schlagen die erste Enklaven-Architektur vor, die mehrere Arten von Enklaven bietet, um die individuellen Anforderungen sensibler Anwendungen zu erfüllen. Cure entwickelt den Stand der Technik bei Enklave-Architekturen weiter und führt neue Hardware-Sicherheitsmechanismen ein, die es erlauben, Systemressourcen, z.B. Prozessorkerne, Cache-Speicher oder Peripheriegeräte, flexibel Enklaven zuzuordnen. Wir entwerfen und implementieren einen Prototyp von Cure für die offene RISC-V-Architektur und evaluieren die Hardware- und Leistungskosten des Prototyps auf einem FPGA- und Simulator-basierten Evaluierungsaufbau.

Auf gerade entstehenden heterogenen Computerplattformen sind potenziell Hunderte von verschiedenen Rechenknoten über eine Network-on-Chip (NoC) Architektur verbunden. Der Einsatz von Enklave-Computing auf NoC-basierten Architekturen bringt viele neue Sicherheitsherausforderungen mit sich, z.B. in Bezug auf die Einbeziehung von Rechenknoten von nicht vertrauenswürdigen Drittanbietern oder die Verteilung sensibler Enklave-Daten über mehrere Knoten hinweg. In unserer Arbeit kombinieren wir das Konzept von Cure mit mehreren Enklave-Typen mit einer neuartigen Hardware-Sicherheitskomponente, dem Distributed Memory Guard (DMG), um die erste Enklave-Architektur für NoC-basierte Plattformen zu entwickeln, welche auch mit nicht vertrauenswürdigen Rechenknoten und verteilten Enklave-Daten umgehen kann.

Dieser Teil der Dissertation basiert auf den folgenden Veröffentlichungen:

[9] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, und Emmanuel Stapf. **CURE: A Security Architecture with CUstomizable and Resilient Enclaves**. In *USENIX Security Symposium*, 2021. CORE Rank A*. Appendix C.

[61] Ghada Dessouky, Mihailo Isakov, Michel A. Kinsy, Pouya Mahmoody, Miguel Mark, Ahmad-Reza Sadeghi, Emmanuel Stapf, und Shaza Zeitouni. **Distributed Memory Guard: Enabling Secure Enclave Computing in NoC-based Architectures**. In *ACM Design Automation Conference (DAC)*, Seiten 985-990, 2021. CORE Rank A. Appendix D.

Neuartige Cache-Designs um vor Seitenkanalangriffen geschützte Enklaven bereitzustellen. Zahlreiche Arbeiten in der Sicherheitsforschung haben die Schwere von Cache-Seitenkanalangriffen auf Computersysteme, einschließlich Enklave-Architekturen, aufgezeigt. Keine der eingesetzten Enklave-Architekturen bietet jedoch Enklaven, welche resistent gegen Cache-Seitenkanalangriffe sind, was ein großes Manko darstellt. Cache-Architekturen, die eine strikte Partitionierung und Zuweisung von Cache-Ressourcen zu Ausführungskontexten ermöglichen, sind ein vielversprechender Ansatz, um starke Seitenkanalsicherheitsgarantien zu bieten. Leider können bestehende Cache-Partitionierungsdesigns entweder nicht alle Angriffe verhindern oder den Cache nur in grobe Partitionen aufteilen, was zu einer unzureichenden Ausnutzung der Cache-Ressourcen führt und es unmöglich macht, die Designs auf eine große Anzahl von Ausführungskontexten zu skalieren. In unserer Arbeit Chunked-Cache schlagen wir eine neuartige Cache-Mikroarchitektur für Enklave-Architekturen vor, die es ermöglicht, Cache-Ressourcen exklusiv und bedarfsgerecht, auf einer feingranularen Cache-Set-Basis, Enklaven zuzuweisen. Auf diese Weise kann Chunked-Cache eine große Anzahl von Enklaven unterstützen und gleichzeitig die gleiche Cache-Verwendungsstrategie bieten, die in herkömmlichen Cache-Architekturen verwendet wird. Wir implementieren einen Prototyp von Chunked-Cache in Hardware und auf einem zyklusakkuraten Cache-Simulator und zeigen in unserer Evaluierung dessen geringe Hardware- und Leistungskosten.

Dieser Teil der Dissertation basiert auf der folgenden Veröffentlichung:

[62] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, und Emmanuel Stapf. **CHUNKED-CACHE: On-Demand and Scalable Cache Isolation for Security Architectures**. In *Symposium on Network and Distributed System Security (NDSS)*, 2022. CORE Rank A*. Appendix E.

Contributions

This dissertation is based on five scientific publications which were made possible through collaborations with excellent researchers and students, which I all thank for their valuable contributions. In the following, I state my contributions to each publication that is part of this dissertation.

Chapter 2 is based on joint works with Ferdinand Brasser, Patrick Jauernig, David Gens and Ahmad-Reza Sadeghi [22], and with Sebastian P. Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi and Thomas Schneider [12]. For Sanctuary [22], I conceived the main ideas and led the research work, whereby Ferdinand Brasser and Patrick Jauernig contributed to the discussions on the design and implementation of Sanctuary. I focused on all aspects of the research project, including the design, implementation and evaluation of the Sanctuary architecture. Ferdinand Brasser contributed to the design of Sanctuary, whereas Patrick Jauernig contributed to the implementation of Sanctuary. Co-author David Gens contributed to the writing of the publication. For Offline Model Guard [12], all co-authors contributed to the discussions on the design and implementation. I focused on the implementation of the keyword recognition algorithm in a Sanctuary enclave and its evaluation. 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 performance evaluation. Patrick Jauernig focused on the design of the communication protocol used by the Sanctuary enclave, the user and the vendor of the machine learning model. Christian Weinert contributed to the discussion of alternative cryptographic secure computation technologies.

Chapter 3 is based on joint works with Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek and Ahmad-Reza Sadeghi [9], and with Ghada Dessouky, Mihailo Isakov, Pouya Mahmoody, Miguel Mark, Shaza Zeitouni, Michel A. Kinsy and Ahmad-Reza Sadeghi [61]. For Cure [9], I conceived the main ideas and led the research work, whereby Patrick Jauernig and Ghada Dessouky contributed to the discussions on the design and implementation of Cure. I 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. Moreover, I supervised the M.Sc. thesis of co-author Matthias Klimmek who focused on the software stack implementation of the Cure architecture and its evaluation. Ghada Dessouky focused on the design, implementation, and evaluation of the cache partitioning for Cure, whereas Patrick Jauernig focused on the implementation of Cure's modifications at the processor and the software stack evaluation. Raad Bahmani and Ferdinand Brasser contributed to the writing of the publication. For Distributed Memory Guard [61], Mihailo Isakov and I conceived the main ideas and led the research work, whereas Miguel Mark and Ghada Dessouky contributed to the discussions on the design and implementation. I focused on the design of the enclave architecture based on the Distributed Memory Guard. Mihailo Isakov focused on the design, implementation and evaluation of the Distributed Memory Guard component. Miguel Mark focused on the analysis of the memory fragmentation problem described in this publication. Shaza Zeitouni, Ghada Dessouky, Pouya Mahmoody and myself contributed to the literature review conducted as part of this research work.

Chapter 4 is based on a joint work with Ghada Dessouky, Alexander Gruler, Pouya Mahmoody and Ahmad-Reza Sadeghi [62]. Ghada Dessouky and myself contributed to the discussions on the Chunked-Cache design and security analysis that resulted in this publication. I focused on the design and implementation of Chunked-Cache on the architectural simulator, its integration, and evaluation. Ghada Dessouky conceived the main ideas and led the research work. She focused on the architecture of Chunked-Cache, its hardware design, implementation and evaluation, and led the evaluation based on the architectural simulator. Alexander Gruler contributed to the benchmarking on the architectural simulator. Pouya Mahmoody focused on the architectural simulator setup and configuration for integrating and evaluating Chunked-Cache.

Acknowledgments

After five years, an exciting but also labor-intensive doctoral journey, which involved victories but also one or the other setback, has finally come to an end. Even though doing one's doctorate is a deeply personal quest, it is almost never possible without the support of many other people. Thus, also in my case there are many people that I would like to thank for supporting me in the last five years. Firstly, I would like to thank my advisor Prof. Ahmad-Reza Sadeghi who agreed to supervise me even though I came to him as an external doctoral candidate which meant that I could at the beginning only spent part of my time in the office of the System Security Lab in Darmstadt. In this context, I would also like to thank all colleagues of the System Security Lab which let me feel welcome in the group. Early on, Ahmad gave me the opportunity to set the focus of my research work and to follow my own research ideas. Ahmad taught me how to build a convincing story around my research results and how to write high-quality research publications which in my experience is one of the most important skills that a researcher has to learn in order to publish his work in the very competitive top-tier conferences of security research.

Getting my research projects done and published would of course not have been possible without the hard work of my co-authors. Thus, my thanks also go to my colleagues at the System Security Lab and external collaborators which were willing to help me implement my research ideas which required dedication and endurance. I also would like to thank Prof. Asokan who agreed to act as my second doctoral referee and the defense committee members Prof. Carsten Binnig, Prof. Reiner Hähnle and Dr. Jean-Paul Degabriele for the interesting discussions during my dissertation defense.

Finally, I would like to thank my family which consistently supported me throughout this journey even though the path that I had chosen might not have been fully understandable for everyone. A special thanks goes to my girlfriend Anki who accepted missing out on many evenings and weekends together over the last years and who gave me the support I needed to bring this fascinating and stirring doctoral journey to a successful end.

Contents

1 Introduction

The protection of sensitive data is of great importance on nearly all computing systems today, whether those are resource-constraint microcontrollers and embedded systems, mobile devices, personal computers or cloud servers. The sensitive data processed by these computing systems range from security- and privacy-sensitive data of individuals (e.g., financial or biometric data), to intellectual property of companies and even sensitive governmental data. Trends such as the Internet of Things (IoT) and new applications (e.g., autonomous driving) and cloud computing services (e.g., Machine-Learning-as-a-Service) even increase the demand for security solutions to protect sensitive data across applications and platforms.

In the last one-and-a-half decades, enclave computing evolved into a very active field of research and also in practice, technologies have been widely deployed to enable enclave architectures, most notably ARM TrustZone-A [151] and TrustZone-M [149], Intel Software Guard Extensions (SGX) [167, 94, 45], AMD Secure Encrypted Virtualization (SEV) [105, 106, 107] and IBM Protected Execution Facility (PEF) [98], whereby also new still undeployed technologies, namely Intel Trust Domain Extensions (TDX) [49] and ARM Confidential Compute Architecture (CCA) [152] have been announced recently. Enclave architectures gained prominence since they guarantee a level of security that goes way beyond the protection capabilities of commodity operating systems. In the following, we first introduce the high-level design of enclave architectures and delimit them from other hardware-assisted security solutions. Then, we provide a categorization for the broad research field of enclave computing. We connect the high-level design of enclave architectures with the corresponding research subfields in Figure 1, whereby we mark defensive research in blue and attack research in orange. Lastly, we define the goals of this dissertation and summarize the contributions that each chapter of this dissertation makes.

1.1 Enclave Architecture Design

One of the key design elements of an enclave architecture are hardware-assisted security mechanisms which are deeply integrated into the System-on-Chip (SoC) and which are configured by a small trusted software component or microcode, which together form the Trusted Computing Base (TCB) of the system. Using the security mechanisms, enclave architectures set up execution contexts, called *enclaves*, that are strongly isolated from each other and all system software (e.g., the operating system or hypervisor), as indicated by the bold lines in Figure 1. Besides the configuration of the security mech-

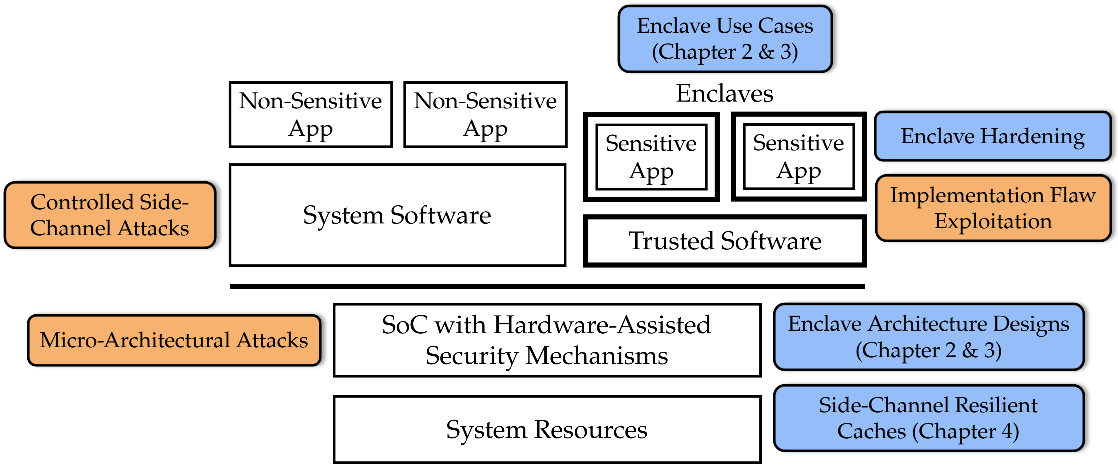

Enclave Use Cases (Chapter 2 & 3)
Enclaves
Non-Sensitive App
Non-Sensitive App
Sensitive App
Sensitive App
Enclave Hardening
Controlled Side-Channel Attacks
System Software
Implementation Flaw Exploitation
Trusted Software
Micro-Architectural Attacks
SoC with Hardware-Assisted Security Mechanisms
Enclave Architecture Designs (Chapter 2 & 3)
System Resources
Side-Channel Resilient Caches (Chapter 4)

Figure 1: Abstract enclave architecture design and subfields of enclave computing research, whereby defensive research is marked in blue and attack research in orange.

anisms, the trusted software also provides cryptographic primitives, e.g. to create keys or perform attestation of enclaves. One key argument for enclave architectures comes from the deliberation that hardware-assisted security mechanisms and a small trusted software component are less likely to contain software-exploitable vulnerabilities than the large code base of commodity system software (e.g., operating systems kernels). Between 2010 and 2020, on average 67 vulnerabilities with a high or critical severity (7.0-10.0 CVSS score [109]) were found in the Linux kernel every year [181]. The clear separation between the commodity system software and the specialized trusted software, which only performs security-relevant tasks, distinguishes enclave architectures also from proposals which combine both tasks in one component, e.g., micro-kernel or micro-hypervisor approaches [80, 223, 124, 92, 145].

The integration of the security mechanisms into the SoC allows to utilize the computational power of the system's main processor for the enclave computations which enables support for a much wider range of applications than solutions that are implemented on dedicated security hardware, either as on-chip secure processors [161, 110, 47, 172] or as off-chip hardware, e.g., Trusted Platform Modules (TPMs) [84] or secure elements [108, 117]. Depending on the design of the security mechanisms and the overall enclave architecture, system resources (e.g., memory, cache, processor cores or peripherals) can be exclusively assigned to enclaves.

The protection of the sensitive applications, encapsulated in enclaves, from all non-sensitive applications and system software, represents the main goal of an enclave architecture. Protection in this context means providing integrity guarantees for enclave program code before its execution and inaccessibility during execution, and integrity and confidentiality guarantees for enclave data at all times. In contrast to most security architectures designed for resource-constrained microcontrollers [65, 224, 175, 126, 82, 207], enclave architectures do not configure fixed enclaves during boot but allow an Operating System (OS) to dynamically load enclaves during runtime, such as it is also provided for non-sensitive applications.

In general, enclave architectures focus in their defense mechanisms on a strong software-based adversary who is able to compromise all system software. Most physical attacks which are performed from close proximity to the target, e.g., the physical analysis of execution times [125], electromagnetic emissions [79] or the power consumption [162], as well as physical fault injections attacks [10, 24, 37] are typically not considered, only simpler physical attacks, e.g., cold boot attacks [89] or bus snooping [169]. Denial-of-service attacks which prevent the execution of enclaves are also mostly not considered since an adversary with control over the system software can trivially shut down the complete system. The assumption of a privileged software adversary marks a key difference to capability and in-process isolation systems [74, 222, 241, 251, 262, 57, 269, 208, 234, 36] which use hardware mechanisms to build sandboxes around processes, e.g., by defining fine-grained access rules for software objects such as pointers. In contrast to enclave architectures, the goal of in-process isolation techniques is to prevent an adversary from escaping the sandbox, not a privileged adversary from entering it. Where most sandboxing techniques assume that an adversary already controls an execution context on the system, other hardware-assisted defenses aim to prevent an adversary from compromising an execution context in the first place using memory corruption attacks [176, 1, 58, 146, 147, 56, 199]. These hardening approaches represent orthogonal work which can be used in conjunction with enclave architectures, e.g. to harden the trusted software component.

1.2 Enclave Computing Research Landscape

In the following, we provide a categorization and summary of the research areas in the context of enclave computing which we also depict in Figure 1. We start by introducing the attack research (marked in orange) which reveals vulnerabilities of enclave architectures and over which we give an overview in [59]. Followed by the defense research (marked in blue) which proposes new enclave architectures or extends existing ones, we discuss some of the most well-known enclave architectures also in [115].

1.2.1 Implementation Flaw Exploitation

Typically, because of its minimal size, the trusted software of an enclave architecture is assumed to be correct. However, as shown by security researchers, deployed trusted software also contains vulnerabilities that once exploited allow to compromise the system [200, 76, 77, 213, 214, 263]. Similarly to the trusted software, the software encapsulated in enclaves is also assumed to be correct, though with the reasoning that the internals of an enclave are out-of-scope for the enclave architecture design. However, most enclave architectures dictate the usage of Software Development Kits (SDKs) to implement enclaves for a specific architecture. Research has shown that these SDKs can contain implementation flaws, e.g., memory corruption vulnerabilities or unchecked return values, which can be exploited to extract sensitive data from enclaves [78, 160, 27, 136, 15, 252, 43, 237, 52].

1.2.2 Controlled Side-channel Attacks

Even though system software is explicitly untrusted, it often still plays a supporting role in the enclave management in order to reuse existing functionality and keep the trusted software to a minimum. Enclave computing research on controlled side-channel attacks has shown that the dependency on system software enables attacks in which the adversary infers information about the internal enclave state by exploiting software structures, e.g., page tables [258, 25, 245, 168, 170], interrupt handlers [236, 91, 235, 254], Direct Memory Access (DMA) memory buffers [139] or other critical software structures [93, 141, 142, 256, 140], as side channels.

1.2.3 Micro-architectural Attacks

The integration of enclave architectures into the SoC saves hardware costs and allows enclaves to use the full computational power of a system's main processor. However, from a security perspective, the integration must be looked at with suspicion since it leads to the sharing of microarchitectural resources between the enclaves and the untrusted software. Security research has shown that especially the sharing of cache resources [19, 88, 83, 53, 210, 266] and the speculative execution unit [137, 67, 102, 26, 33, 127, 211, 238, 239, 195] can be exploited to extract sensitive data from enclaves. Moreover, since adversary and victim run on the same processor, software-based fault injection attacks could successfully be demonstrated on enclaves architectures [230, 191, 190, 171, 120].

1.2.4 Enclave Architecture Designs

Apart from the deployed enclave architectures based on ARM TrustZone [151, 149, 152], Intel SGX [167, 94, 45, 49], AMD SEV [105, 106, 107] and IBM PEF [98], the defensive line of enclave computing research proposed a wide range of new enclave architectures to withstand current state-of-the-art attacks. The key components which define the capabilities and limitations of an enclave architecture are the underlying hardware-assisted security mechanisms upon which enclaves are implemented. Thus, enclave architectures can be roughly divided into those that introduce new hardware-based security mechanisms [225, 28, 18, 253, 4, 51, 9, 12, 66, 242, 113, 270, 172, 16, 183, 116, 257, 227] and those that do not require hardware changes to be implemented and instead utilize already existing hardware mechanisms [22, 135, 165, 166, 226, 40, 97, 114, 267, 80, 260, 95, 198, 7, 144, 35, 265, 119, 143, 133, 264], whereby many approaches rely on virtualization technologies [165, 166, 40, 97, 95, 198, 260, 7, 144, 35, 119, 264]. Apart from a categorization depending on the need for hardware modifications, the research that proposes enclave architectures can also be grouped by the processor architecture the work focuses on, e.g., ARM [22, 226, 40, 97, 267, 265, 114, 143, 133], x86 [66, 165, 166, 242, 113, 270, 80, 260, 95, 198, 7, 144, 35, 119, 183, 264] or RISC-V [253, 51, 135, 9, 172, 16].

1.2.5 Side-channel Resilient Caches

Protecting enclaves from cache side-channel attacks is an important research challenge which unfortunately as been mainly ignored in the deployed enclave architectures. Also, only recently, research proposed side-channel resilient caches which explicitly target enclave architectures with their design [60, 62, 202]. Taking the characteristics of enclave computing into consideration is crucial for the cache design, e.g., the peculiarity that the elevated security guarantees provided by enclaves are not required for all applications on the system but only for a sensitive subset must be reflected in the cache design. Since not many caches specifically designed for enclave architectures have been proposed, we also include more generic proposals for side-channel resilient caches. In general, the proposed cache designs can be separated into two major groups. Partitioning-based approaches [60, 62, 202, 247, 123, 248, 259, 64] remove side channels on the cache by assigning cache resources exclusively to execution contexts and by preventing all accesses (read, write and evict) from untrusted software, thus hindering the adversary from gaining enough information about the victim's cache utilization for an attack. In contrast, randomization-based approaches [154, 249, 233, 193, 194, 255, 229, 201] do not assign cache resources exclusively but randomize the mapping from memory addresses to cache sets to prevent an adversary from inferring accessed memory addresses from cache misses, which is crucial for performing a cache side-channel attack.

1.2.6 Enclave Hardening

Apart from designing new enclave architectures that withstand state-of-the-art attacks, other enclave computing research has been concerned with hardening the enclaves of existing enclave architectures by proposing new defense mechanisms against specific attacks. The categorization of these research works can be aligned with the already introduced attack vectors on enclave architectures, namely exploitable implementation flaws, cache side-channel attacks and controlled side-channel attacks. Hardening against implementation flaws is achieved through the formal verification of the enclave code or trusted software [219, 220, 134, 71], type-safe programming languages [243] or by finding vulnerabilities in the enclave code through fuzzing techniques [43]. The defenses against cache side-channel attacks and controlled side-channel attacks mostly rely on randomization techniques, e.g., by adding random noise to memory traces [29], randomizing data locations [21] or the address space of the enclave [212], or they rely on forcing an enclave to only perform oblivious memory accesses [2, 3, 205]. Other works use specialized transactional memory hardware to hide page faults from an attacker [216], detect an abnormal enclave interrupt behavior [34] or lock enclave data completely in the cache to prevent side channels [85]. Pinning enclaves to processor cores [179, 32] is another option to at least make the core-exclusive side channels inaccessible for an adversary.

1.2.7 Enclave Use Cases

In general, enclave architectures do not focus on specific use cases but instead aim to provide isolated execution environments that can cater various applications which process sensitive data. Thus, another line of research explores the applicability and usefulness of enclave architectures for the protection of a variety of sensitive applications, e.g., databases [75, 129, 187], blockchain applications [54, 38], data analysis [209, 268], multiparty computation [68, 8, 131], language runtimes [204], virtual TPMs [196] or compilers [73]. One type of application which has been studied extensively in the context of enclave architectures is machine learning. Different strategies have been proposed how enclaves should be leveraged for their protection, e.g., for privacy-preserving model training on remote servers [104, 100, 231, 178, 232, 185, 138, 20, 192] or on local devices [90, 206, 12], whereby in some works computationally intensive operations are offloaded to Graphics Processing Units (GPUs) to improve performance [232, 185, 101, 206]. Apart from evaluating the applicability of enclaves for specific use cases, other research works take a more general approach and aim to provide solutions for porting unmodified existing applications to enclave architectures [11, 30, 5, 217, 215, 99, 86] which is challenging since some architectures, such as Intel SGX, heavily restrict system calls for its enclaves.

1.3 Dissertation Goals

The goal of this dissertation is to significantly contribute to enclave computing research by tackling multiple research challenges which we describe in the following.

Enclave Architecture Designs. The goal of enclave architectures to make our computing systems generally more secure requires that all sensitive applications with elevated security demands can be protected in enclaves. Unfortunately, enclave architectures build on the TrustZone technology, which are the most deployed enclave architectures today, suffer from design limitations which make it impossible to provide TrustZone-based protection to third-party applications without security concerns. Existing research works are either based on virtualization techniques which blocks them for their intended usage [40, 97, 114], rely on a weak isolation between enclaves based on virtual address spaces [267, 143, 133, 265], or only provide temporal isolation which is highly unpractical [226]. One goal of this dissertation is to develop new concepts to enforce a strong hardware-assisted separation (spatial and temporal) between enclaves based on the TrustZone technology so that an unrestricted access to TrustZone's protection capabilities can be provided to all applications without putting the security of the computing system in danger.

Apart from limitations specific to the TrustZone technology, most enclave architectures lack mechanisms to enable a secure communication between enclaves and commodity peripherals, which is required to support uses cases in which sensitive sensor data is processed, e.g. biometric authentication, or when sensitive graphic computations are offloaded to a GPU. The enclave architectures which are capable of secure I/O in general either cannot assign peripherals directly and exclusively to single enclaves [40, 226, 97, 267, 265, 114, 143, 133, 198], require peripheral modifications [119], rely on encrypted communication between the enclave and the peripheral [242, 113] or are focused on off-chip GPUs in data centers [270]. In this dissertation, another goal is to develop new mechanisms to enable secure communication channels between enclaves and peripherals which support commodity peripherals and do not rely on encryption schemes. Moreover, another goal is to adapt the enclave architecture designs also to emerging NoC-based computing platforms on which the presence of untrusted computing nodes and the distributed enclave data present additional challenges.

Side-channel Resilient Caches. Cache side-channel attacks are a persistent threat also to enclave architectures. For their protection, side-channel resilient cache designs are required which take the peculiarities of enclave architectures into account, e.g., the assumption of a privileged adversary or the differentiation between applications that require protection and those that do not. So far, the majority of the proposed cache architectures [247, 123, 248, 259, 203, 154, 249, 233, 193, 194, 255, 229, 201] do not target enclave architectures. Moreover, many of the proposed designs do not scale because they can assign cache resources only in a coarse-grained fashion [123, 247, 248] or cannot protect against stealthier occupancy-based cache side-channel attacks [154, 233, 229, 64, 193, 194, 249, 255, 201, 60, 259]. In this dissertation, our goal is to develop scalable cache designs focusing on enclave architectures which provide protection against side-channel attacks (including occupancy-based attacks [218]) in the presence of strong software adversaries and which minimize their performance impact on non-sensitive applications and system software on the platform.

Enclave Use Cases. One of the main use cases for enclave architectures frequently discussed are machine learning applications. One reason is that these applications handle different types of sensitive data which must be protected, namely privacy-sensitive user data, integrity-sensitive machine learning algorithms and machine learning models which represent intellectual property. So far, most works focused on a scenario in which the machine learning applications run on servers in the public cloud [104, 100, 231, 178, 232, 185, 138, 20, 192]. However, today most mobile devices perform machine learning tasks using privacy-sensitive user data collected locally on the device, e.g., speech recognition/processing, biometric authentication or video/image processing. Thus, another goal of this thesis is to investigate how enclave architectures can be utilized to protect user input and machine learning algorithms and models on mobile devices.

A widespread usage of enclave architectures in the future will heavily depend on which types of application the next generation of enclave architectures will support. Currently, enclave architectures are static concerning the type of application they can effectively protect, depending on the assumptions that were made during design time. Their static nature increases the overhead for porting existing applications to enclaves or makes the protection of certain applications not possible without security concerns. In order to ease the deployment of unmodified applications in enclaves, researchers proposed multiple approaches to extend enclave capabilities, e.g., by secure file or network I/O or multithreading [11, 30, 5, 217, 215, 99, 86]. Unfortunately, all proposals require cumbersome workarounds for the inherent limitations of the underlying enclave architecture. One important goal of this dissertation is to develop a new enclave architecture design which does not make assumptions regarding the applications which are protected in its enclaves. Instead, the architecture should be flexible enough to protect any type of application appropriately without workarounds, e.g., if a sensitive application heavily relies on system calls, the architecture must securely provide them without relying on error-prone defense mechanisms, such as the verification of system call returns [31].

1.4 Dissertation Outline

In the following, we summarize the remainder of this dissertation, whereby each chapter describes our contributions and compares them to related work.

Chapter 2: We present SANCTUARY [22], a novel enclave architecture which utilizes the TrustZone technology to provide strongly-isolated enclaves on ARM-based systems without relying on virtualization. SANCTUARY runs sensitive applications de-privileged on temporarily isolated physical processor cores by making use of TrustZone's versatile address-space controller. This enforces a two-way hardware-level isolation in which sensitive applications are protected from compromised system software while the system is also protected from potentially malicious applications in the enclaves. Thus, additional sensitive applications do not increase the attack surface of the system which allows to make the TrustZone technology openly accessible without security concerns. We implement SANCTUARY on a multi-core ARM-based development board and demonstrate its practicality by thoroughly evaluating our prototype.

Moreover, we present Offline Model Guard (OMG) [12], which represents a real-world use case implementation on the SANCTUARY platform. We show that SANCTUARY enclaves can be used to perform offline keyword recognition using TensorFlow Lite for microcontrollers on mobile devices while guaranteeing the privacy of the user's speech data, the secrecy of the machine learning models provided by the service provider, and the integrity of the machine learning algorithms.

Chapter 3: We present CURE [9], the first enclave architecture which offers multiple different types of enclaves to protect diverse sensitive applications. CURE provides sub-space enclaves to enable intra-privilege-level isolation, user-space enclaves with a small memory and TCB footprint, and also kernel-space enclaves which provide an isolated runtime and drivers to the sensitive applications. CURE introduces new hardware-assisted security mechanisms which allow kernel-space enclaves to securely communicate with commodity peripherals over Memory Mapped Input/Output (MMIO) and DMA, without relying on encrypted communication streams. Moreover, CURE's design includes a way-based cache partitioning scheme to protect enclaves from cache side-channel attacks. We implement Cure for the open RISC-V architecture and thoroughly evaluate our prototype in terms of hardware and performance overhead on an evaluation setup consisting of ISA simulators, a cycle-accurate system simulator and an FPGA. On standard benchmarks, Cure incurs a geometric mean performance overhead of 15.33%.

Additionally, we present the Distributed Memory Guard (DMG) [61], a novel hardware security component which allows to design enclave architectures for heterogeneous computing platforms that connect a large number of on-chip computing nodes over an NoC architecture. We investigate the problem of memory fragmentation on systems under heavy load and propose an enclave architecture design based on the DMG which can handle distributed enclave data and untrusted computing nodes.

Chapter 4: We present Chunked-Cache [62], a novel side-channel resilient cache design targeting enclaves architectures. Chunked-Cache's design is based on a strict cache partitioning strategy which assigns cache resources on a cache-set granularity exclusively to enclaves. Thus, Chunked-Cache scales to a large number of enclaves and at the same time minimizes the performance impact on the system software and all non-sensitive applications. The strict partitioning of Chunked-Cache allows to also prevent occupancy-based cache side-channel attacks. We implement a hardware model of Chunked-Cache and include it in a cycle-accurate cache simulator. We evaluate the hardware and power consumption overhead of our Chunked-Cache prototype and provide a thorough performance evaluation using real-world applications and standard computing benchmarks, which shows that Chunked-Cache induces on average a 43% lower cache miss rate than way-based partitioning schemes.

In **Chapter 5**, we conclude this dissertation and propose directions for future research on enclave computing.

2 Openly-accessible Enclave Computing on ARM-based Devices

The ARM TrustZone technology was one of the first set of hardware-assisted security mechanisms which allowed to implement enclave architectures. The traditional enclave architectures based on TrustZone, which are now widely deployed on mobile devices, partition the system into two *worlds*, the *normal world* and the *secure world*. The normal world contains the commodity untrusted Operating System (OS) and all non-sensitive applications, whereas the secure world contains a Trusted OS (TOS) which isolates sensitive applications, called Trusted Apps (TAs), using separate virtual address spaces. Moreover, the TOS provides all OS services to the TAs, such as memory management, thread handling and also the handling of sensitive peripherals, e.g., fingerprint sensors or iris scanners. The enhanced functionality of a typical TOS leads to a large code basis in the order of 100 KLOC [27]. As a result, the TOS presents a large attack surface, especially for the TAs which communicate with the TOS over a feature-rich interface. Thus, every TA that is added to the secure world increases the probability of vulnerabilities that might lead to a compromise of the system [200, 76, 77, 213, 214, 263, 78]. Applying our definition of an enclave architecture presented in Section 1.1, all software running in the secure world must be viewed as a coherent enclave and the traditional architectures based on TrustZone thus as single-enclave architectures.

The mobile device vendors mostly share this view on TrustZone and thus security-minded vendors either prevent third-party developers completely from deploying own apps in the secure world, or only allow TAs which passed a rigorous and costly certification process. As a result, the security capabilities of TrustZone are mostly used to protect services of the device vendors, e.g., key management or Digital Rights Management (DRM), whereas many applications which process privacy-sensitive user data, e.g. machine learning applications which collected data over the mobile phone camera or microphone, remain unprotected. To change the current situation, new TrustZone-based multi-enclave architectures are required which can protect additional sensitive applications without an increase of the system TCB and thus enable an open access to TrustZone's protection capabilities.

In this chapter, we describe our contributions to TrustZone-based enclave architectures and to the protection of machine learning applications using enclaves (Section 2.1). Moreover, we summarize related research and position our work to it (Section 2.2).

2.1 Contributions

In this section, we summarize the contributions of this dissertation to providing an openly-accessible TrustZone and in showcasing the protection of machine learning applications in enclaves.

2.1.1 SANCTUARY: ARMing TrustZone with User-space Enclaves

This dissertation contributes to the enclave computing research of *Enclave Architecture Designs* (Section 1.2.4) by designing, implementing and evaluating a novel TrustZone-based enclave architecture, named Sanctuary, which is described in the following publication that can be found in Appendix A:

[22] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **SANCTUARY: ARMing TrustZone with User-space Enclaves**. In *Symposium on Network and Distributed System Security (NDSS)*, 2019. CORE Rank A*. Appendix A.

Normal-world Enclaves. The core idea of Sanctuary is to cut the direct connection between the TCB size of TrustZone-based architectures and the number of protected sensitive applications by constructing strongly-isolated enclaves in the normal world. The *de-privileged* Sanctuary enclaves, shown in Figure 2, comprise a sensitive application and a runtime which provides all required OS services to the sensitive application, e.g., memory management, thread or interrupt handling. Thus, the code base running in the secure world can be substantially reduced and in the Sanctuary design, it only contains security-relevant code from the device vendor which primarily manages the setup of the Sanctuary enclaves. The secure world code, called security primitives, can also offer some security services to the enclaves if required, such as remote attestation or sealed storage. Thus, in contrast to the traditional TrustZone-based architectures, the secure world presents only a minimal interface, or no interface at all, to the sensitive applications encapsulated in Sanctuary enclaves. As a result, a malicious or compromised enclave does not pose a threat to the overall system security which in turn enables an open access to TrustZone since all third-party developers can freely protect their sensitive applications within Sanctuary enclaves.

Sanctuary's normal-world enclaves are constructed by making use of a so far overlooked feature of the TrustZone-enabled memory controller provided by ARM, called the TrustZone Address-Space Controller (TZASC) [148]. The TZASC is typically used to assign memory regions to either the normal or secure world which is achieved by performing access control on all memory transactions the TZASC receives over the system bus. The differentiation between normal and secure world transactions is made based on the Non-Secure (NS) signal which is part of every memory transaction and which indicates whether the sender of the transaction, e.g. a processor core, is currently executing in the normal or secure world mode. The world mode, and thus the value of the NS sig-

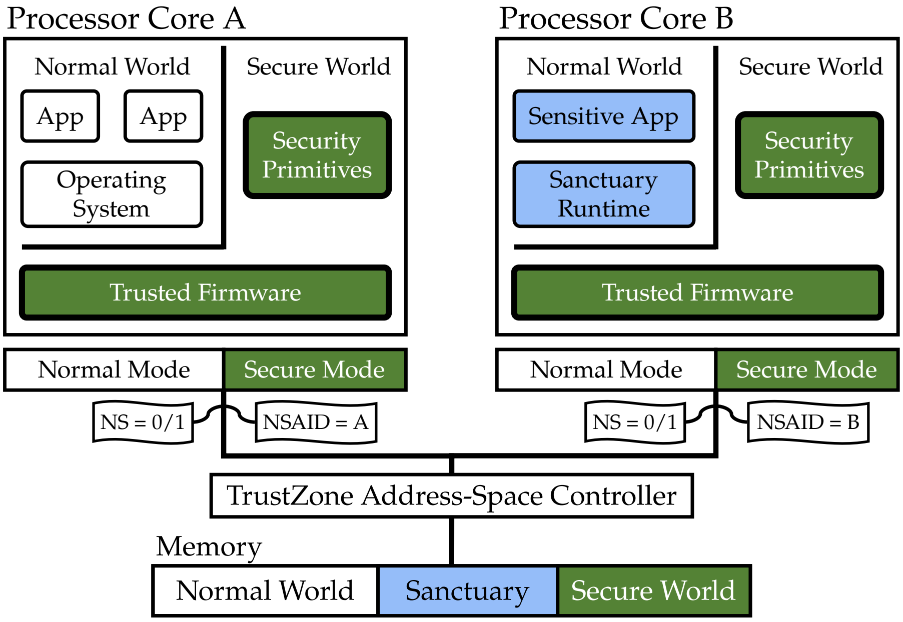


Figure 2: High-level design of the Sanctuary enclave architecture which isolates the Sanctuary enclave, consisting of sensitive application and runtime, on processor core B. The TCB marked in green consists of the trusted firmware and security primitives provided by the device vendor.

nal, is configured by the trusted firmware during context switching between the normal and secure world. The trusted firmware forms the highest-privileged software running on TrustZone-enabled processors. As we discovered during our research, the TZASC is also able to perform access control upon another bus signal, the Non-Secure Access ID (NSAID) signal, which is used to identify senders of memory transactions. As depicted in Figure 2 in Sanctuary, we assign an unique NSAID to every processor core on the system and use the TZASC to bind normal-world memory regions temporarily to single physical processor cores. With this isolation primitive, we can construct Sanctuary enclaves in the normal world.

The setup of a Sanctuary enclave begins with the request of a non-sensitive application to execute a sensitive application. Subsequently, the commodity OS loads the sensitive application binary and Sanctuary runtime, and identifies the processor core with the least amount of load. Next, the selected core is freed from all pending processes and suspended. After a switch to the secure world, the trusted security primitives configure the TZASC in a way that the memory region containing the sensitive application and Sanctuary runtime is made accessible exclusively for the suspended core. After verifying the application and runtime binaries using digital signatures, the security primitives wake up the suspended core which starts executing the Sanctuary runtime. After the runtime is finished booting, it jumps into the sensitive application code. The sensitive application can now perform its requested task, whereby Sanctuary enables a communication over shared memory with the untrusted OS and the security primitives in the secure world. When the sensitive application finished its execution, the security primitives securely store the enclave state, clean the enclave's memory, reconfigure the TZASC and give control over the core back to the OS. While an enclave is running, all sensitive data stored in the core-exclusive cache structures, e.g., the Translation Lookaside Buffer (TLB) or L1 cache, are automatically protected from a malicious OS since Sanctuary enclaves are isolated on physical processor cores. Regarding the shared L2 cache, Sanctuary can enforce that no enclave data gets cached in it.

Implementation & Evaluation. We implement a prototype of Sanctuary on the HiKey 960 development board containing the Huawei Kirin 960 chip set frequently used on mobile devices. We implement a normal-world OS based on the Linux kernel and include a custom kernel module to enable the communication with the Sanctuary enclave and secure world. We implement the security primitives provided by the device vendor based on a reduced version of OP-TEE [153]. For the Sanctuary enclave runtime, we use a modified version of the Zircon microkernel [158] which also offers a user space in which we run the sensitive applications. Sanctuary's design does not require hardware modifications, only the NSAID assignment must be altered to achieve that every processor core receives an unique ID which makes them distinguishable for the TZASC.

We evaluate the performance of Sanctuary using microbenchmarks and a two-factor authentication use case. In our microbenchmark evaluation, we measure basic operations performed during the life cycle of an enclave, including setting up the enclave on an isolated core, communicating with the normal and secure world, and tearing down the enclave. For the enclave setup, we measure around 200 ms (450 ms when excluding all enclave data from the L2 cache), whereby most of the time (59%) is spent on shutting down the selected processor core for which we rely on the hotplugging mechanism provided by the Linux kernel. Booting the Zircon microkernel constitutes for another 30% of the enclave setup time. Most of the remaining time is used for loading and verifying the Sanctuary binaries. For the enclave shutdown, we measure a duration of around 100 ms, whereby the cleaning of the enclave memory constitutes for 45% of the time and the core reboot for 53%. We also measure the time required to call a Sanctuary enclave from the normal world and also how long it takes to call the secure world from a Sanctuary enclave. For the former case, we measure 150 us on average, whereas for the later case we measure 310 us on average. The reason for the time doubling is that in our prototype, a call from the Sanctuary enclave to the secure world is performed over the normal world and thus two context switches are required. This can be prevented by including the secure-world driver also in the Sanctuary enclave runtime.

For our two-factor authentication use case, we implement an One-Time Password (OTP) generator in a Sanctuary enclave. In a first step, the enclave performs, with the help of the security primitives, remote attestation at a backend to receive a secret key which, in a second step, is used by the enclave to generate valid OTPs. In our evaluation, we measure a duration of 1.2 s (1.8 s without L2 cache) for the complete process from the start of the secret key provisioning up to the point where the user receives the OTP. For the key provisioning step, which is only performed once after installation, we measure 884 ms on average (1174 ms without L2) and for the OTP generation step 365 ms on average (630 ms without L2). Our use-case evaluation shows that Sanctuary provides a practical performance. Moreover, Sanctuary does not negatively influence the user experience since the OS remains fully responsive during the enclave execution.

2.1.2 Offline Model Guard: Secure and Private ML on Mobile Devices

This dissertation contributes to the enclave computing research of *Enclave Use Cases* (Section 1.2.7) by designing, implementing and evaluating a secure and private offline machine learning service, called Offline Model Guard (OMG), using the Sanctuary enclave architecture. OMG is described in the following publication that can be found in Appendix B:

[12] 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 *Conference on Design, Automation and Test in Europe (DATE)*, pages 460-465, 2020. CORE Rank B. Appendix B.

Protocol Design. The Offline Model Guard (OMG) service is based on the Sanctuary enclave architecture [22] and protects sensitive Machine Learning (ML) applications in an offline scenario on mobile devices. The protection of an ML application in its entirety is a challenging task since it requires to protect different types of sensitive data. Firstly, the privacy-sensitive user data which the ML algorithm processes, e.g. voice data, must be securely collected on the device without being intercepted by an adversary. Thus, the device must provide a secure connection between the ML application and the used sensor. Secondly, the ML model used for inference, which is typically trained in a backend system on large data sets, must be securely provisioned to the device since the model presents valuable intellectual property for the vendor of the ML application. Lastly, the integrity of the ML algorithm must be guaranteed at all times to prevent a manipulation of the ML algorithm which could lead to false inference results.

OMG provides privacy, secrecy and integrity for ML applications. We depict OMGs design in Figure 3. In order to protect a sensitive ML application with OMG, the application is first implemented in a Sanctuary enclave and bundled with a runtime that supports all libraries required by the application, e.g., a specific ML framework. Then, the enclave is provisioned to a system implementing the Sanctuary enclave architecture. When an application of the commodity OS triggers the execution of the sensitive ML application, a corresponding Sanctuary enclave is created, shown in blue in Figure 3. Then after enclave setup ①, the integrity of the ML algorithm is proven to the OS application and the ML vendor using the attestation functionalities of Sanctuary. Next ②, the enclave establishes an encrypted communication channel to the backend of the ML vendor which is used to provision the enclave with an ML model. Subsequently ③, the received encrypted and rollback-protected ML model is stored outside of the enclave for later use. The provisioning and storing of the ML model must only be performed once when the

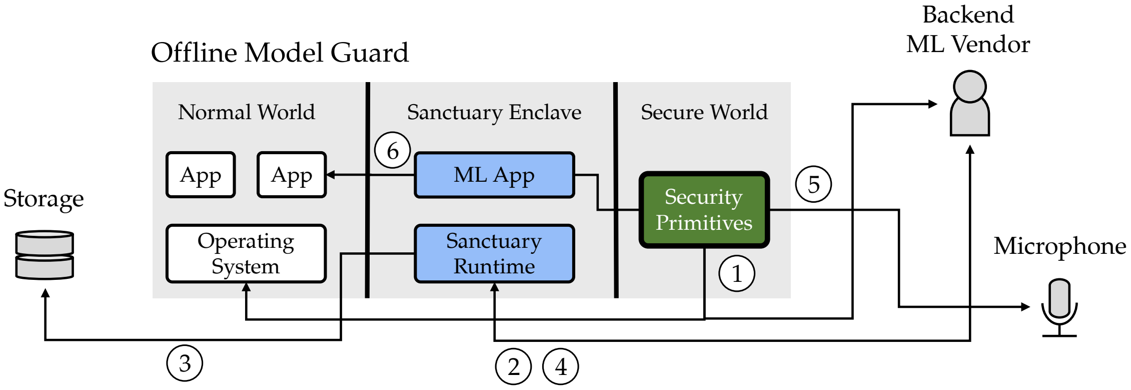


Figure 3: Offline Model Guard (OMG) design using the Sanctuary enclave architecture which splits the system into the normal world, secure world and Sanctuary enclaves. OMG provides privacy for the user input, secrecy for the ML model and integrity for the ML algorithm.

ML application is used for the first time, or, when the model is updated by the ML vendor. OMG reuses the OS functionality for storing the ML model and communicating with the ML vendor backend which is not a security concern since all data is encrypted. We omit these details in Figure 3 for the sake of clarity. To use the ML model, the enclave again establishes a secure connection to the vendor to receive a decryption key for decrypting the model ④. Then, the enclave sets up a secure connection to a sensor, e.g. a microphone, over the secure world software ⑤, marked in green, and starts collecting user data. After the data collection, the ML algorithm performs inference upon the data. The classification result is then returned to the OS application ⑥. The OMG design also allows to perform offline training with the collected data on the mobile device, depending on whether the computational power of the mobile device application processor is sufficient for this task.

Implementation & Evaluation. We implement OMG on a HiKey 960 development board extended with the Sanctuary enclave architecture. In our prototype, we implement an offline keyword recognition algorithm in an enclave which can classify command keywords when receiving an audio file as an input. As the ML algorithm, we use a small neural network, consisting of a 2D convolutional layer, a ReLU activation layer and a mapping layer, which we implement using the TensorFlow Lite for microcontrollers library [157]. We perform the model training on a separate device using the Speech Command [250] data set consisting of 105,000 WAVE audio files of people saying 30 different commands, whereby we preprocess the audio files and generate a compressed spectrogram from each audio signal. The resulting compressed model, which we load into our keyword recognition enclave, has a size of 49 KB. In our performance evaluation, we measure the time for a keyword inference performed in a Sanctuary enclave and compare it against an execution outside of an enclave. The setup of the enclave takes around 200 ms until completion, including the suspension of a processor core, the verification of the enclave and the runtime boot. During the ML inference, Sanctuary introduces only a small performance overhead of 2% which is caused by the context switching between the enclave, normal and secure world. We also verify that the classification accuracy of the ML algorithm is not influenced by Sanctuary. Our evaluation demonstrates that OMG is capable of providing privacy, secrecy and integrity for sensitive ML applications on mobile devices at moderate performance costs.

2.2 Related Work

In the following, we summarize research related to Sanctuary [22] and OMG [12], and compare it to our work. For Sanctuary, we focus on the most related work, namely enclave architectures which target ARM-based computing systems. For a more general summary of enclaves architectures, we refer the reader to Section 3.2.1. For OMG, we focus on related work which protects machine learning applications in enclaves.

2.2.1 Enclave Architectures for ARM-based Computing Systems

We categorize the enclave architectures focusing or ARM-based platforms by the technologies that are used to isolate enclaves from each other and the system software, namely virtualization technologies, software such as an operating system kernel, and TrustZone's hardware mechanisms.

Virtualization-based Isolation

Typically, hardware-assisted virtualization technologies introduce a high-privileged software component, called hypervisor, which manages the hardware resources invisible to the Virtual Machines (VMs). Because of its elevated rights, security research frequently proposed to use the hypervisor to construct enclave architectures.

On ARM-based systems, Cho et al. [40] propose a design in which the hypervisor is used to create a software compartment that is isolated from the normal world and secure world software, called the On-demand Software Protection (OSP) world. Whenever the execution of a sensitive application, called Security Critical Code (SCC) by the authors, is triggered from the normal world software, the virtualization extensions are activated on all processor cores by making use of Inter-Processor Interrupts (IPIs). Then, the OSP hypervisor is loaded from the secure world to setup the OSP world. Next, the SCC binary is decrypted, loaded into the OSP world and then executed in user space. When the SCC terminates, the OSP world is stored in the secure world and the virtualization extensions turned off. As a result, OSP only suffers from performance degradation caused by the second-level address translation while an SCC is executed. However, it is crucial for the design that the virtualization extensions are activated on all cores before an SCC is executed since a privileged adversary could otherwise map the memory of an SCC into his own memory space. To prevent a compromised core from ignoring the IPIs, the configuration of the General Interrupt Controller (GIC) is performed during the secure boot process and locked afterwards. The authors of PrivateZone [114] propose a design very similar to OSP. Also in PrivateZone, a third isolated compartment next to the normal and secure world is set up, called Private Execution Environment (PrEE), in which the sensitive applications are executed, shielded from the normal world. In contrast to OSP, the trusted software which configures the PrEE is not executed in the hypervisor mode but in monitor mode which is the highest privileged software level on an ARM processor.

In vTZ [97], virtualization technologies are also utilized to design an ARM-based enclave architecture, whereby an enclave is represented by a hardened VM running in the normal world. In contrast to OSP [40] and PrivateZone [114], vTZ runs a complete Xen hypervisor [103] on the platform which contradicts the enclave architecture design goal of a minimal software TCB. Thus, the authors propose to outsource all sensitive functionality, e.g., the VM and hypervisor memory management or the handling of peripherals and VM states, from the hypervisor to the secure world in order to remove the hypervisor from the TCB. Whenever a context switch between VMs is performed or when a VM's page tables must be modified, a switch to the secure world is required to verify the decisions of the hypervisor. Moreover, to make sure that a compromised hypervisor is not circumventing the security mechanisms of vTZ, the hypervisor code must by analyzed to verify that it does not contain any sensitive instructions, e.g., to disable the Memory Management Unit (MMU) or to modify the page table base address.

Software-based Isolation

The traditional TrustZone-based enclave architectures assume a secure OS running in the secure world which isolates enclaves, represented by Trusted Apps in user space. Since the security of all enclaves depends on the correctness of the relatively large secure OS (in the order of 100 KLOC [27]), we refer to this isolation mechanism as software-based. The size of a typical secure OS grew larger over time since more and more functionality was added to it, e.g., the support of secure peripherals such as fingerprint sensors. In early works on TrustZone, the size of the secure world code was kept small by using only a trusted language runtime instead of a full OS to manage the Trusted Apps [204]. Comparable designs have also been proposed for the M-Shield security technology from Texas Instruments [128]. Multiple newer works try to strengthen enclave security by again reducing the software TCB from the complete secure OS to a smaller subset. TEEv [143] substitutes the secure OS with a hypervisor-like component, called TEE-visor, which manages enclaves in the form of VMs. Since hardware-assisted virtualization in the secure world is not yet available, the authors of TEEv propose other security mechanisms to separate the kernel of an enclave VM from the TEE-visor which both run in the secure privileged exception level (S-EL1). To achieve this, the authors rely on intra-privilege-level isolation techniques proposed by related work [55, 41]. In TEEv, all memory management is performed by the TEE-visor which sets up different address spaces for the enclaves and itself. Additionally, a gating component is required in the secure world which administers the secure context switches from enclaves to the TEE-visor. As in vTZ [97], TEEv must verify that no enclave contains sensitive instructions which could circumvent TEEv's security mechanisms. PrOS [133] also provides a software-only virtualization for the secure world which suffers from the same limitations as TEEv. However, in contrast to TEEv, the trusted software in PrOS managing the enclaves, called PVisor, is executed in the monitor mode (EL3). Thus, a virtualized secure OS can occupy the S-EL0 and S-EL1 privilege levels. TEEv and PrOS both rely on intra-privilege-level separation techniques to isolate enclaves in the form of VMs. With the ARMv8.4 architecture revision, ARM introduces hardware-assisted virtualization also in the secure world which allows to achieve the goals of TEEv and PrOS more efficiently without workarounds. However, to this date, Apple's A13-A15 are the only processors that already implement the ARMv8.4 architecture specification.

Some works proposing enclave architectures adopt the software-based enclave isolation of traditional TrustZone-based architectures and instead focus on other limitations of TrustZone. The authors of CaSE [265] focus on the inability of TrustZone to protect sensitive data in the Dynamic Random Access Memory (DRAM) from physical attacks, such as cold-boot attacks [89] or bus snooping [169]. It must be noted though that a successful attack on a DRAM chip connected to an ARM-based SoC represents an advanced physical attack, which requires costly technical equipment, since the DRAM chips are typically soldered on top of the SoC in a Package-on-Package (PoP) fashion which makes them hard to reach for an adversary. In CaSE, all sensitive applications are encrypted when stored in the DRAM. When a sensitive application is invoked, CaSE's trusted software, which runs in the secure world, loads the encrypted binary in the L2 cache and marks all cache lines with the Non-Secure (NS) flag so that only the secure world can access it. Then, CaSE verifies and decrypts the application binary. During run-time of the application, the L2 cache is locked so that no sensitive data can be evicted from the cache by an adversary. On every context switch between applications or when performing memory paging, CaSE encrypts the application data before it leaves the SoC. SecTEE [267] achieves a similar goal by executing the complete secure world software in an on-chip memory instead of the cache. Additionally, SecTEE considers the attack class of cache side-channel attacks. The authors propose a page-coloring scheme which assigns secure world memory to the sensitive applications in such a way that they do not compete over the same cache sets. Similar to CaSE, SecTEE uses the cache locking mechanism of ARM processors to prevent an adversary from evicting sensitive application data from the cache.

Temporal TrustZone-based Isolation

In TrustICE [226], the TrustZone hardware mechanisms which separate normal and secure world are also used to protect sensitive applications from each other and the normal-world OS in so-called Isolated Computing Environments (ICE). When the normal-world OS is executed, all ICE instances are stored in the secure-world memory. When an ICE execution is triggered from a normal-world application, the secure-world software, called Trusted Domain Controller (TDC), verifies the binary of the called ICE, suspends the complete normal-world OS and reconfigures the secure-world memory partition achieving that the ICE binary becomes part of the normal-world memory. In the following, control is given to the ICE which runs as a standalone component in the normal world. When the ICE execution is finished, the TDC clears the processor state and caches, reconfigures the memory partitions and restores the normal-world OS. Thereby, TrustICE achieves a temporal isolation of the sensitive applications from the OS since always just one of both is executed on the system at one point in time.

Comparison

The main goal of SANCTUARY is to overcome the single-enclave limitation of traditional TrustZone-based architectures and instead provide a design with multiple enclaves that are strongly isolated using TrustZone's hardware-assisted security mechanisms. This goal marks a key difference to the works CaSE [265] and SecTEE [267] which rely on the software-based isolation of Trusted Apps used in traditional TrustZone-based architectures which was shown to be insufficient [27]. The protection from physical attacks on the memory chip, on which CaSE and SecTEE focus, is an orthogonal problem. The proposed solutions, locking the sensitive applications in the cache or in on-chip memory, are also applicable to SANCTUARY. SecTEE is the only related work which also considers cache side-channel attacks, whereby SecTEE implements a page-coloring scheme which requires modifications at the memory management software of the Trusted OS. In contrast, SANCTUARY protects the enclaves by excluding all enclave memory from the shared cache. TEEv [143] and PrOS [133] also rely on a software-based isolation but use it to implement software-based virtualization in the secure world. The main limitation of both approaches is that they require to check all enclave binaries for instructions that could circumvent the isolation which is impractical and error-prone. SANCTUARY does not need to resort to techniques for intra-privilege-level isolation and instead provides a strong isolation on temporarily separated physical processor cores.

Another design goal of SANCTUARY is to not rely on the hypervisor privilege level for managing the enclaves and configuring the hardware-assisted security mechanisms, since this blocks the hypervisor for its intended purpose, namely hardware virtualization, or for other usages required by the device vendor, e.g., for running firmware code. In contrast to OSP [40], PrivateZone [114] and vTZ [97], SANCTUARY does not deploy any software in the hypervisor privilege level. Therefore, SANCTUARY does not induce a performance overhead caused by second-level address translation from which all works relying on virtualization [40, 114, 97] suffer to some extent. Moreover, to protect against DMA attacks, virtualization-based approaches require Input/Output Memory Management Units (IOMMUs) in front of all DMA-capable devices. In SANCTUARY, the enclave memory is automatically protected from rogue peripheral accesses. Comparable to TEEv [143] and PrOS [133], vTZ [97] must analyze the hypervisor binaries to prevent the inclusion of privileged instructions, e.g. to modify the page table base address, which could be used to bypass the security mechanisms of the enclave architecture.

The key difference to TrustICE [226] is that SANCTUARY does not require to suspend the OS when an enclave is executed. SANCTUARY provides spatial and temporal isolation for enclaves, whereas TrustICE can only provide temporal isolation which makes this approach highly limited in practice since a suspended OS is completely unresponsive to input from the device user. Moreover, TrustICE only provides a one-way isolation which means that the OS cannot access the enclave data but a malicious enclaves might be able to access data of the suspended OS since it is stored in the normal world memory. Sanctuary in contrast provides a two-way isolation where the enclaves are protected from a malicious OS and, at the same time, the OS from malicious enclaves.

2.2.2 Protecting Machine Learning Applications in Enclaves

Ohrimenko et al. [178] were one of the first to protect ML applications in SGX enclaves. The authors assume a scenario where sensitive data from multiple data providers is locally encrypted and aggregated on a remote server to train ML models, whereby SGX enclaves are used to protect the unencrypted data during the training process. After the training, the models are shared with all data providers to perform inference upon them. In this scenario, the SGX enclaves might leak information to the untrusted system software on the server through data-dependent access patterns [258]. Therefore, the authors develop data-oblivious variants of standard ML algorithms, e.g., support vector machines, neural networks or decision trees, which guarantee that all accesses from the ML algorithms to the memory, disk or network do not depend on secret data.

Myelin [104] provides security guarantees comparable to the work from Ohrimenko et al. [178] since it relies on data-oblivious implementations of Deep Neural Networks (DNNs). In Myelin, every model owner compiles its deep learning model into a privacy-preserving model graph which is then trained on a remote server inside of an SGX enclave, using privacy-sensitive training data. Privado [231] also considers the leakage of memory access patterns when DNNs are executed in SGX enclaves on a remote server. The authors show that this leakage can be exploited by an adversary to learn which classification result of the network corresponds to which neuron activation when doing inference. This is possible since the activation of a neuron depends on a threshold function which performs data-dependent memory accesses. When a Machine-Learning-as-a-service (MLaas) customer sends encrypted data to the enclave, the adversary can collect memory access patterns to infer the classification of the encrypted data. As Ohrimenko et al. [178], the authors of Privado propose modified ML algorithms which are free of input-dependent access patterns. Occlumency [138], as Privado [231], focuses on deep-learning inference as a use case and proposes optimization techniques to speed up the inference in SGX enclaves.

In Chiron [100], an MLaas scenario is considered where input data is collected from an MLaas customer and used to train ML models without revealing the data to the MLaas provider or the ML algorithms to the MLaas customer. This is achieved by performing the model training in a Ryoan [99] enclave (based on SGX) which protects the MLaas customer's data but still offers the MLaas provider the possibility to freely select, configure and train the ML models. SecureTF [192] has the same goal as Chiron and provides a general ML framework for SGX which is based on Scone [5]. MLCapsule [90] follows a different approach and protects the customer data by never sending it to the cloud. Instead, the ML algorithms and models are sent to the customer and protected from direct customer access in SGX, thus MLCapsule supports an offline MLaas scenario.

VoiceGuard [20] targets the use case of speech recognition. In VoiceGuard, sensitive audio data is collected from user devices, e.g., IoT devices like Amazon Echo, Google Home or Apple HomePod, and sent to a service provider where the data is used for inference on proprietary ML models coming from ML vendors. The service provider shields the computations in an SGX enclave, thereby protecting the privacy-sensitive user data and the proprietary ML models of the ML vendors. The inference results are then sent back to the user devices.

All approaches described so far execute the ML algorithms on the main processor implementing SGX. However, many state-of-the-art ML algorithms use GPUs (or other hardware accelerators) to achieve a higher performance [130]. Slalom [232] also utilizes SGX enclaves but offloads parts of the ML computations to GPUs. The authors idea is to split the computation of DNNs in sensitive and non-sensitive parts, whereby only the sensitive parts are executed on the main processor in an SGX enclave. The non-sensitive parts are accelerated on a GPU. Using SGX, Slalom achieves protection for the privacy-sensitive user data and integrity for the ML computations, but not confidentiality for the ML model. The authors of eNNclave [206] also split the ML computations into sensitive and non-sensitive parts. However, in contrast to Slalom [232] which offloads all linear layers of the DNN to the GPU, eNNclave offloads only the last dense layers in which it keeps all confidential model parameters. Moreover, eNNclave assumes an offline inference scenario on the client side comparable to MLCapsule [90].

In Telekine [101], a scenario is assumed in which cloud GPUs are rented to boost the performance of locally executed ML algorithms. The authors assume a GPU architecture that provides enclaves, as proposed by Graviton [242], which enables an encrypted communication between the trusted client and the GPU. Even though all data streams are encrypted, the authors show that in this scenario an adversary can predict the classification of encrypted input data by observing the timing of the GPU kernel execution. Telekine proposes a client library which modifies the commands send to the GPU in a way that all data streams become data oblivious. Visor [185] also relies on GPU enclaves as provided by Graviton [242], however in Visor, the authors assume that the ML application, in this case Video-analytics-as-a-service, is completely executed on the cloud server, whereby the video decoding and image processing is performed in the SGX enclave and the classification on the GPU.

Comparison

One key difference between Offline Model Guard (OMG) [12] and most related work is that OMG focuses on protecting the inference of sensitive ML applications when performed on local user devices. The model training, which requires large data sets and consumes a lot of computational resources, is assumed to be performed on the trusted backend of the ML vendor. Ohrimenko et al. [178], Myelin [104], Chiron [100], SecureTF [192] and Slalom [232] all focus on protecting privacy-sensitive user data during the ML model training on remote servers. Privado [231], VoiceGuard [20], Occlumency [138] and Visor [185] also target remote servers but focus on ML inference. The only other works that focus on ML inference on local user devices are MLCapsule [90] and eNNclave [206].

Apart from the usage scenario, all other works rely on Intel SGX for protection, whereas some execute the ML algorithm on the main processor [178, 104, 100, 192, 231, 20, 138, 90], others offload security uncritical tasks to GPUs [232, 206] or assume an enclave architecture on the GPU to also offload security-sensitive computations [101, 185]. OMG, in contrast, utilizes Sanctuary enclaves and thus is the only approach directly applicable to mobile devices. From a security perspective, relying on Sanctuary enclaves also has other advantages. Firstly, it allows OMG to protect the ML applications from controlled side-channel attacks, without modifying the ML algorithms, since Sanctuary enclaves contain a runtime that provides OS services independently from the untrusted commodity OS. Secondly, OMG is the only work that is able to tackle the challenge of securely collecting privacy-sensitive user data on the local user device. OMG achieves this by using the TrustZone secure world for privacy-preserving data collection, which, in contrast to Intel SGX, enables a secure communication between enclaves and sensors, e.g. a microphone. MLCapsule [90] and eNNclave [206], which also focus on offline inference, do not consider the challenge of secure data collection.

3 Flexible Enclaves Across Platforms

The requirements sensitive applications impose on a security architecture vary considerably between applications, e.g., whether they require dynamic memory allocation, multithreading, secure file or network I/O, a secure user interface or secure communication channels to peripherals, such as sensors or specialized computing hardware. In the enclave computing landscape, various enclave architectures have been proposed which all provide a certain *type* of enclave, depending on what the architecture designers thought would be the best enclave to protect sensitive applications running on the system. As a result, many architectures [35, 260, 95, 198, 166, 45, 66, 51, 18, 18, 253, 4] provide enclaves which comprise only a process (or part of a process) in user space which offers a small memory footprint and a minimal TCB. However, user-space enclaves typically rely on services provided by the untrusted OS which makes them susceptible for side-channel attacks. Other architectures [116, 264, 227, 257, 97, 105, 49, 152] encapsulate complete VMs in enclaves which makes them suitable for protecting existing software stacks without modifications but similar to user-space enclaves, VM enclaves depend on services from the untrusted system software, in this case the hypervisor. In contrast, still other architectures [165, 7, 135, 151, 265, 267, 22, 226, 133, 40, 114] provide enclaves which are largely self-sustained and do not depend on the untrusted system software which makes them more resilient against attacks from a compromised OS or hypervisor and allows them to provide a secure interaction with peripherals. The independence from the system software, however, increases the memory footprint and TCB size of these enclaves, which we call kernel-space enclaves, which makes them more susceptible to implementation flaws.

All enclave types have their pros and cons and are better or worse suited to protect a particular sensitive application. Nevertheless, existing architectures follow a *one-size-fits-all* approach in which all sensitive applications must be protected within a single type of enclave. In order to satisfy the requirements of a diverse set of sensitive applications and support more enclave use cases, without building software workarounds to extend one-type enclave architectures [11, 30, 5, 217, 215, 99, 86], new designs are required which provide more flexible enclaves so that the security architectures adapt to the requirements of the sensitive applications and not the other way around. This becomes even more important in emerging heterogeneous computing platforms where computing nodes, e.g. processors or hardware accelerators, are connected over a Network-on-Chip (NoC) architecture. In such a scenario, enclaves must deal with a potentially large number of different computing nodes with which sensitive applications might demand to securely interact.

In this chapter, we describe our contributions to enable flexible enclaves across platforms (Section 3.1), summarize related research and position our work to it (Section 3.2).

3.1 Contributions

In this section, we summarize the contributions of this dissertation to providing flexible enclaves on computing platforms with traditional bus architectures and emerging Network-on-Chip (NoC) architectures.

3.1.1 CURE: A Security Architecture with CUstomizable and Resilient Enclaves

This dissertation contributes to the enclave computing research of *Enclave Architecture Designs* (Section 1.2.4) by designing, implementing and evaluating a novel enclave architecture, named Cure. Being the first architecture which provides multiple types of enclaves on one platform, Cure also contributes to the enclave computing research of *Enclave Use Cases* (Section 1.2.7). Cure is described in the following publication that can be found in Appendix C:

[9] 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 *USENIX Security Symposium*, 2021. CORE Rank A*. Appendix C.

Hardware-assisted Security Mechanisms. Cure's high-level design is depicted in Figure 4, whereby all hardware modifications are colored in blue. At its core, Cure's design is based on a system bus extended with additional registers and control logic to implement access control capabilities. On every port of the system bus that connects to a child component, e.g., a peripheral or the memory bus, registers and control logic are added after the system bus arbitration logic, whereas at the ports connecting parent components, e.g. DMA-capable devices, the additional hardware is introduced before the address decoder. In combination, the added registers and logic form a system-wide access control mechanism at a central location, named filter engine in Figure 4, which observes all bus transactions, whether they come from the processor cores and target peripherals or the memory controller, or from DMA-capable devices, e.g. a GPU connected over a DMA controller. In Figure 4, we leave out DMA-capable devices for the sake of clarity. In Cure, this central access control mechanism is used to assign contiguous memory regions to enclaves. Moreover, it enables an enclave-to-peripheral binding in which, on the one hand, MMIO peripherals can be exclusively assigned to enclaves, and, on the other hand, also DMA-capable devices, configured over MMIO, can be bound to enclaves. In both cases, Cure enables a non-encrypted communication with the enclaves.

Apart from the modifications at the system bus, Cure requires small hardware changes at the processor cores. Cure introduces an additional register at the highest-privileged

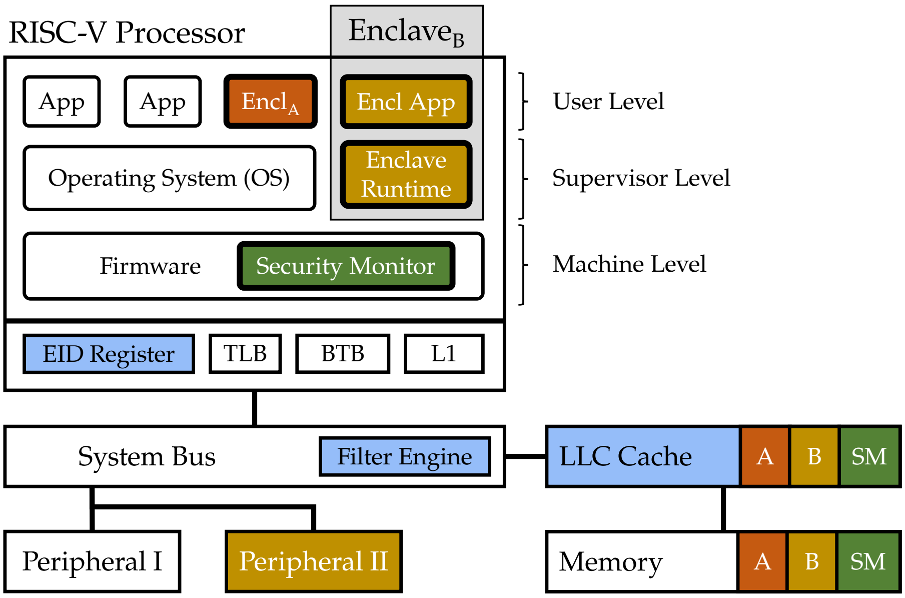


Figure 4: High-level design of the Cure enclave architecture, providing user-space enclaves (Enclave$_A$), kernel-space enclaves (Enclave$_B$) and sub-space enclaves (represented by the Security Monitor). Modified hardware components are marked in blue.

software level, named the Enclave ID (EID) register, that indicates which enclave a processor core currently executes. Further modifications enforce that the EID register can only be accessed by the TCB of the system, which we call Security Monitor (SM), and which runs in the highest-privileged software level. The EID register is used as a system-wide identifier for the currently executed enclave. Thus, to perform access control upon it, the EID content must be transmitted to Cure's hardware-assisted security mechanisms as part of every memory transaction. In Cure, this is achieved by extending the bus protocol, which is implemented between the processor core, cache and system bus, with an EID signal. Moreover, Cure's design also includes a modified Last-Level Cache (LLC) which tags cache line with the ID of the owning enclave and which allows to assign cache ways to enclaves, on-demand and adjustable during runtime, in order to protect them from cache side-channel attacks. The configuration of all security mechanisms is performed by the SM which also provides security services to the enclaves, such as local and remote attestation, key management or a secure rollback-protected storage for the enclave state.

Enclave Types. The described hardware-assisted security mechanisms enable Cure to provide multiple types of enclaves to protect diverse sensitive applications. First of all, since Cure's security mechanisms are located at the system bus outside of the processor boundary, access control is performed on physical memory addresses. This allows Cure to also enforce access control for bare-metal software which in turn enables Cure to support self-sustained kernel-space enclaves (Enclave$_B$ in Figure 4) which execute a privileged runtime inside the enclave boundary that can provide services, such as memory management, multi-threading or interrupt handling to the enclave, independent from the untrusted OS. The assignment of memory regions, cache resources or peripherals to an enclave is performed by the SM when setting up the enclave by configuring the hardware-assisted security mechanisms and modified cache controller accordingly. By assigning peripherals to enclaves and by including the corresponding device drivers into the enclave runtime, Cure achieves an enclave-to-peripheral binding also for unmodified peripherals which do not support encrypted communication.

Besides providing kernel-space enclaves, Cure uses the same security mechanisms to also provide user-space enclaves (Enclave$_A$ in Figure 4) with a small TCB which are scheduled by the OS like regular unprotected applications. Since user-space enclaves depend on the services provided by the OS, Cure protects the enclaves from page-based controlled side-channel attacks [258, 25, 245, 168, 170] by storing all enclave memory pages in the enclave memory. The untrusted OS is still in charge of the enclave memory management, however, on every OS request to modify enclave page tables, a switch to the SM is performed which verifies the management decisions of the OS to prevent an overlap between the memory regions of enclaves and the OS. Moreover, to protect from interrupt-based controlled side-channel attacks [236, 91, 235, 254], Cure enables user-space enclaves to register interrupt trap handlers at the SM. Whenever an enclave is interrupted and the interrupt handled by the OS, the SM jumps into the registered trap handler when re-entering the enclave allowing the enclave to implement their own defense mechanisms based on heuristics to detect ongoing attacks by an abnormal interrupt behavior. In order to prevent cache side-channel attacks on the core-exclusive cache structures, the SM performs a flush of the L1 cache, TLB and Branch Target Buffer (BTB) on all context switches in and out of a user-space enclave.

Lastly, Cure's security mechanisms can also be used to setup sub-space enclaves which represent execution contexts isolated on an intra-privilege level. Sub-space enclaves are especially useful in a scenario in which the SM shares the highest-privileged software level with other software components, such as platform firmware or software which emulates missing hardware features. By using Cure's security mechanisms to run the SM as an sub-space enclave (Security Monitor in Figure 4), the TCB of the system can be reduced considerably.

Implementation & Evaluation. We develop a Cure prototype for the open RISC-V architecture using the Rocket Chip generator [6] in which we include our proposed hardware-assisted security mechanisms. Cure's software components mainly comprise of the SM, which consists of only 3 KLOC, the kernel-space enclave runtime which we implement based on a minimal Linux kernel, and a kernel-module to enable the OS to interact with the SM during enclave setup. Moreover, some modifications at the interrupt and memory management code of the OS, which we also base on a Linux kernel, are performed to support user-space enclaves.

We evaluate the hardware overhead induced by our prototype in terms of Lookup Tables (LUTs), the fundamental programmable logic blocks of FPGA devices, and registers by synthesizing our hardware model for a Virtex UltraScale FPGA device and by comparing the results to an unmodified minimal baseline RISC-V system with 2 Rocket cores. The hardware-assisted security mechanisms introduced in front of the memory bus cause an overhead of 8.6% for the LUTs and 3.8% for the registers. Moreover, for the hardware added to perform access control on memory transactions targeting peripherals, a LUT and register increase of 0.4% must be accepted for every protected peripheral. For also controlling DMA-capable devices, 0.2% increase of LUTs and 0.3% increase of registers must be invested for every DMA-capable device present in the system. The extension of the bus protocol with the EID signal adds another 0.4% of hardware overhead to the LUTs and registers. The proposed way-based partitioned LLC results in 1.7% overhead for the LUTs and 1.8% for the registers, compared to an unmodified LLC. Overall, our evaluation shows that CURE only introduces small hardware overheads.

Apart from the hardware overhead, we also analyze the performance overhead introduced by CURE. The access control mechanisms added to the system bus introduce no additional cycle latency since all access control is performed in the same cycle which we verify using the cycle-accurate system simulator Verilator [221]. Moreover, we use Verilator to measure the flushing operations on the cache resources performed during all enclave context switches. For running benchmarks on our CURE prototype, we use the ISA simulators Spike [177] and QEMU [13] which we enhance by the cycle overheads introduced by the flushing operations. We then split our performance evaluation into microbenchmarks and macrobenchmarks. In the microbenchmark evaluation, we measure the overhead of basic enclave operations and evaluate the difference between user-space and kernel-space enclaves. In general, our results show that for an enclave setup, most of the setup time, 91.3% for the user-space enclave and 52.1% for the kernel-space enclave, is spent on the cryptographic verification of the enclave binary. Moreover, our evaluation demonstrates the increased boot time of the kernel-space enclave which is caused by its bigger code base and the enclave runtime boot. During the teardown of the user-space and kernel-space enclave, 39.9% and 45.7% of the time are spent on zeroing the enclave memory, respectively. In our macrobenchmark evaluation, we run the rv8 [42] and CoreMark [63] benchmarks inside of user- and kernel-space enclaves and compare our results to an execution outside of an enclave. On average, the kernel-space enclave induces a performance overhead of 15.33%, whereas the user-space enclave causes 19.70% overhead which shows that kernel-space enclaves quickly compensate the additional overhead they introduce during the enclave setup since they introduce no performance overhead on context switches from the user to kernel space since the switches are performed inside of the enclave boundary. In general, our performance evaluation shows that CURE only introduces a moderate performance overhead on applications encapsulated in enclaves.

3.1.2 Distributed Memory Guard: Enabling Secure Enclave Computing in NoC-based Architectures

This dissertation contributes to the enclave computing research of *Enclave Architecture Designs* (Section 1.2.4) by designing and implementing a novel hardware security component, called the Distributed Memory Guard (DMG), and by designing a DMG-based enclave architecture for NoC-based platforms. The DMG and the enclave architecture based on it are described in the following publication that can be found in Appendix D:

[61] Ghada Dessouky, Mihailo Isakov, Michel A. Kinsy, Pouya Mahmoody, Miguel Mark, Ahmad-Reza Sadeghi, Emmanuel Stapf, and Shaza Zeitouni. **Distributed Memory Guard: Enabling Secure Enclave Computing in NoC-based Architectures**. In *ACM Design Automation Conference (DAC)*, pages 985-990, 2021. CORE Rank A. Appendix D.

Distributed Memory Guard. On NoC-based platforms, security mechanisms are typically proposed to be implemented in the Network Interfaces (NIs), which connect the computing and memory nodes to a network of routers, since the NIs observe every transaction entering or leaving the network like gatekeepers. Thus, an NI can intervene in the network communication by rerouting or blocking transactions. When an NI is used to perform access control on memory transactions, an important question is on which granularity access policies must be defined to provide a practical security mechanism. In this publication, we answer this question by analyzing the degree of physical memory fragmentation on NoC-based platforms. For this, we execute the long-running cloud service benchmark suite CloudSuite [182] on an Intel server which represents a typical usage scenario for NoC architectures since cloud servers tend to comprise a large number of processor cores. Our results show that even when only utilizing up to 70% of the server's available memory, a high memory fragmentation occurs. After running the benchmarks for 60 minutes, the biggest free contiguous memory region on the system is only 32 KB in size. Moreover, defragementation of the system cannot be achieved sufficiently by killing single benchmarks. Instead, the complete server has to be rebooted. On an NoC-based platform, the fragmented memory regions are additionally distributed among the memory nodes. Our evaluation demonstrates that to support the memory consumption behavior of commodity cloud services, access control mechanisms must provide fine-grained policies at least at a per-page (4 KB) granularity.

One approach to provide fine-grained access control is to integrate MMU-like hardware components working on page table hierarchies into each NI. However, MMUs introduce a large hardware overhead and traversing multiple levels of page tables upon a TLB miss adds a significant performance overhead to memory requests. Thus, for the Distributed Memory Guard (DMG), which we locate between the computing node and NI, we follow another approach to provide a per-page access control mechanism with low hardware and performance overhead. In particular, we propose to perform access control based on rules inspired by Wash-Hadamard Transforms (WHT) which decompose a discrete binary vector into a superposition of basis functions called Walsh functions. We use this mechanism to construct access control rules which match a pattern of free pages in the physical memory. By combining multiple rules, a set of memory pages in a fragmented memory space can be covered. The DMG uses these pattern-based access rules, stored in a rule table protected in secure memory, to assign memory regions to execution contexts on a per-page basis. In hardware, the DMG consists of a rule cache which caches the

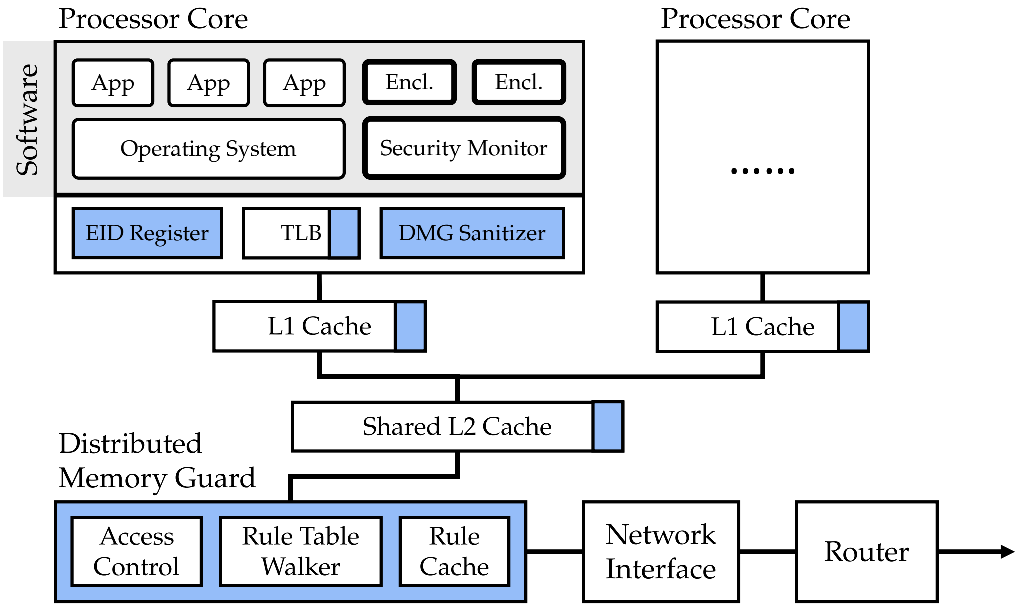


Figure 5: Enclave architecture design based on the Distributed Memory Guard. Modified hardware components are marked in blue, whereby for the L1 and L2 caches only a 1-bit tag is added for each cache line and for the TLB, a tag with the EID of the enclave which requested the address translation is added to each entry.

most frequently used rules, access control logic to compare the rules with the observed memory transactions, and a rule table walker which searches the (single-level) rule table when no fitting rule is found in the rule cache. On a rule hit in the cache or rule table, the memory transaction is allowed to enter the network, whereas on a rule miss in the cache and rule table, the transaction is rejected and an interrupt triggered by the DMG to inform the software about the access violation.

DMG-based Enclave Architecture. Based on the Distributed Memory Guard, we design the first enclave architecture, shown in Figure 5, which targets NoC-based platforms. Introducing a DMG in front of every NI allows to connect also unmodifiable and potentially untrusted IP-blocks from third-party providers to the network. However, it is not enough to build an enclave architecture since the DMG cannot perform access control inside of a computing node and thus cannot protect sensitive applications from a potentially malicious or compromised OS running on the same node, which is a basic characteristic of an enclave architecture (Section 1.1).

In order to create a functioning enclave architecture, we adopt the multi-enclave concept from Cure [9] and adapt it to the NoC scenario. We introduce a new hardware enclave ID (EID) register at the highest-privileged software layer which tracks the currently running enclave and which is only accessible for the Security Monitor (SM), the trusted software of the platform. The SM is responsible for creating the pattern-based access rules and configuring all DMGs, so that memory regions are exclusively assigned to enclaves running across the computing nodes. Since the DMGs observe all memory transactions on the network, whether they target memory addresses in the main memory, MMIO regions of peripherals or DMA regions, the SM can set up secure communication channels between enclaves and peripherals without relying on encrypted communication streams. Moreover, as shown in Figure 5, we add hardware logic, called the DMG sanitizer, at all processor cores of the modifiable computing nodes which is identical to the access control logic implemented in the DMG. On every memory access, after the address translation from virtual to physical address, the MMU uses the DMG sanitizer to check the memory request against the pattern-based rules in the rule cache. Analog to the DMG, on a cache miss, a page table walk is performed to find a fitting rule in the rule table. The verified translations stored in the TLB are then tagged with the EID of the requesting enclave. If no rule is found, an interrupt is triggered which is handled by the SM. By performing access control directly in the MMU, memory requests hitting the cache can be verified. Nevertheless, a 1-bit tag must be introduced at every cache line to indicate whether a cache line is occupied by enclave data. This is required since a malicious OS might be able to disable address translation completely which would allow the adversary to directly access data in the cache.

In combination, the DMGs in front of the NIs and the hardware modifications at the processor cores allow to construct an enclave architecture which on the one hand enables fine-grained enclaves inside of the computing nodes, and, on the other hand prevents illegal accesses from untrusted third-party IP blocks connected to the network.

3.2 Related Work

In the following, we summarize research related to Cure and DMG, and compare it to our work. For Cure, we discuss enclave architectures across computing platforms. For DMG, we summarize security research focusing on heterogeneous computing platforms.

3.2.1 Enclave Architectures Across Platforms

We categorize enclave architectures depending on the type of enclaves that the architecture provides, namely enclaves comprising a process (or part of a process) in user-space, enclaves which comprise a complete VM or self-sustained kernel-space enclaves which run privileged software and do not depend on the system software for management functionalities. We choose this categorization since the provided enclave type defines many of the capabilities and limitations of an enclave architecture.

User-space Enclaves

One of the pioneer works which provides enclave-like execution environments running in user space is AEGIS [225]. In AEGIS, a concept for a security-enhanced single processor is introduced which allows to run sensitive applications either in Tamper-Evident (TE) or Private and authenticated Tamper-Resistant (PTR) execution environments. The

TE provides integrity for the sensitive application by verifying its initial program state, the program code and data in caches and off-chip memory, and by preserving the program register state over interrupts. The PTR additionally provides privacy for the program code and data by clearing the processor registers during context switches and by encrypting all off-chip memory using a hardware encryption engine.

Many works propose to use virtualization technologies to isolate a piece of sensitive software from an untrusted OS. Overshadow [35] modifies the VMware Virtual Machine Monitor (VMM) to create mappings of one guest physical memory page to multiple host physical memory pages, called *multi-shadowing*, whereby one of the host physical memory pages is encrypted and integrity protected. In contrast to AEGIS [225], the cryptographic operations are performed by the VMM instead of a hardware engine. Overshadow then isolates sensitive applications, called cloaked applications, from an untrusted guest OS by presenting the OS an encrypted view on the physical memory pages assigned to the cloaked applications, whereas a normal unencrypted view is presented to the cloaked application. Each cloaked application is wrapped by a shim layer which interposes all communication between the application and the OS. SP^3 [260] and InkTag[95] follow an approach similar to Overshadow [35] and modify the Xen [228] and KVM [132] hypervisors respectively, in this way intercepting memory accesses to encrypt or decrypt memory pages depending on the permissions of the requesting execution context. AppSec [198] also utilizes a VMM to protect sensitive applications from an untrusted OS via nested page tables. However, in contrast to Overshadow [35], SP^3 [260] and InkTag[95], AppSec uses a small custom hypervisor instead of a modified commodity hypervisor and does not encrypt the memory pages of the sensitive applications. Instead, AppSec interposes every memory access of the OS on sensitive application memory, e.g. when returning from a system call, and verifies if the memory access is aligned with the sensitive application's access rules defined by the parameters of the issued system call. Another work utilizing virtualization technologies is TrustVisor [166]. In TrustVisor, the authors propose an enclave architecture design based on a small trusted hypervisor to overcome the performance problems of their earlier work Flicker [165] which suspends the OS while running sensitive code. TrustVisor separates the sensitive code, called Pieces of Application Logic (PAL), from the OS during run-time using nested page tables. Thus, the OS does not need to be suspended. A Root of Trust for Measurement (RTM) is provided on the platform by combining a hardware TPM to measure the TrustVisor component with software-based TPMs, included in TrustVisor, to measure every booted PAL. Minibox [144] builds on TrustVisor but includes a service runtime in the isolated PAL to handle some system calls, e.g. for dynamic memory management, directly in the PAL. To protect the OS from a potentially malicious or compromised PAL, Minibox utilizes the sandboxing solution NaCl [262].

Inspired by the research on security architectures, Intel developed the Software Guard Extensions (SGX) [45, 167, 94] which provide isolated execution environments for sensitive applications for which Intel coined the term *enclaves*. In SGX, each enclave comes bundled with a non-sensitive application that invokes the enclave as one of its child processes. Thus, enclave and host process share the same virtual address space. The enclave code and data is protected from an unauthorized access by the system software through hardware-assisted security mechanisms. SGX is mostly implemented through microcode inside of the processor, e.g., by introducing a new set of instructions to communicate with the enclaves and control them. Additionally, hardware changes are made at the Page Table Walker (PTW). Whenever a TLB miss causes a page fault, the SGX microcode performs access control upon the virtual address by checking whether the address points to enclave memory and whether the current execution context is allowed to access the enclave memory. Only when the access is permitted, the address translation is performed and the physical address loaded into the TLB. In a recent work, Park et al. [183] show how the access control mechanism of SGX can also be extended to provide a hierarchy of enclaves where multiple *inner* enclaves share one *outer* enclave. SGX also provides a remote attestation mechanism for enclaves which is implemented in a set of enclaves provided by Intel, e.g. the Quoting Enclave which computes attestation signatures. In SGX, the untrusted OS performs the enclave memory management, handles the enclave exceptions and provides I/O services to the enclaves. All enclave code or data leaving the processor is encrypted by a hardware component called the Memory Encryption Engine (MEE), comparable to the hardware engine proposed in AEGIS [225], which allows SGX to protect enclaves from simple hardware attacks [89, 169]. The research proposal Iso-X [66], implemented on the OpenRISC platform, aims to overcome SGX's limitation that the enclave memory can only be allocated in a small memory region exclusively carved out for enclave usage during boot time. To achieve this, the authors introduce a new set of processor instructions and extend a commodity MMU by a new PTW which allows to flexibly assign memory regions to enclaves using per-enclave page tables. Another work which resembles SGX's and Iso-X's [66] high-level concept is Sanctum [51]. One key goal of Sanctum, which was implemented on the open RISC-V architecture, is to protect enclaves from a range of side-channel attacks SGX was shown to be vulnerable to. In Sanctum, the untrusted OS still manages the enclave memory and provides OS services to the enclaves, e.g. I/O services or interrupt handling. However, Sanctum stores the enclave page tables in the enclave memory to prevent controlled side-channel attacks which target page tables [258, 25, 245, 168]. Moreover, enclaves in Sanctum are interrupt-aware which allows them to implement security policies that can detect ongoing controlled side-channel attacks targeting interrupts [236, 91, 235]. Additionally, Sanctum implements memory page coloring as a defense against cache side-channel attacks [19, 88, 83, 53, 210] which allows to assign cache sets of the shared LLC exclusively to enclaves. Sanctum enforces the isolation of the enclave code and data by introducing small hardware changes at the PTW of the processor. Comparable to SGX, the circuitry added around the PTW prevents a successful address translation of virtual memory addresses which would map to physical memory addresses that the current execution context is not allowed to access. In contrast to SGX, the security critical functionalities of Sanctum are not implemented in microcode but mainly in a software component running in the most privileged software level of a RISC-V processor, the machine level. MI6 [16] extends the isolation guarantees of Sanctum [51] to speculative out-of-order processors. One key goal of MI6 is to provide enclaves resilient against transient-execution attacks [137, 67, 102, 26, 33, 127, 211, 238, 239, 195] which is mainly achieved with speculation barriers and a *purge* instruction which on context switches scrubs all microarchitectural resources that might leak information of an enclave's state. Moreover, MI6 prevents also minor leakages between execution contexts running on separate cores which share the LLC microarchitectural structures, e.g., by partitioning the Miss Status Handling Registers (MSHR) or by adding a round-robin arbiter in front of the cache-access pipeline.

Other works which provide enclaves in user space focus on resource-constraint microcontrollers. One of the first works, TyTAN [18] introduces a new exception engine and extends the execution-aware Memory Protection Unit (MPU), introduced in the predecessor work TrustLite [126], to isolate sensitive tasks in enclaves from an untrusted real-time OS. In contrast to TrustLite which focuses on remote attestation, TyTAN allows the dynamic loading of enclaves during run-time, whereas the execution-aware MPU configuration is only static for the TCB of the system. Moreover, TyTAN which was implemented on the Intel Siskiyou Peak embedded platform, provides secure communication between enclaves and real-time guarantees. The authors of TIMBER-V [253] present another enclave architecture targeting microcontrollers. In TIMBER-V, which was implemented on the open RISC-V architecture, memory tagging is combined with an extended MPU to provide intra-privilege-level isolation in the user level and supervisor (kernel) level. In the user level, TIMBER-V provides a fine-grained separation of tasks into a sensitive and a non-sensitive part, whereby the sensitive parts (enclaves) can only be entered over a fixed set of entry points. In the supervisor level, TIMBER-V uses the same mechanism to isolate a software component called the TagRoot from the rest of the untrusted OS. The TagRoot is responsible for configuring the memory tags of the enclaves in the user level, using newly introduced tagging instructions, and provides OS functionalities to them, e.g., interrupt handling. AION [4], which builds on previous security architectures for microcontrollers [175], combines real-time guarantees with availability guarantees by introducing a new hardware exception engine. In contrast, RT-TEE [244] provides real-time guarantees for TrustZone-based security architectures on microcontrollers by introducing a policy-based event driven hierarchical scheduler.

Virtual Machine Enclaves

Using the hypervisor (or VMM) for isolating sensitive applications from an untrusted OS is proposed by multiple research works [35, 260, 95, 198, 166, 144]. In all these proposals, the hypervisor is *misused* in the sense that it is not utilized to abstract hardware resources for multiple OS but for separating and protecting user-space enclaves from a single untrusted OS. In another line of research [116, 264, 227, 257], enclave architectures are proposed in which the hypervisor is used for its intended usage, namely hardware virtualization. By introducing new hardware-assisted security mechanisms, these proposals provide enclaves, in the form of complete VMs, protected from a potentially malicious or compromised hypervisor. The authors of H-SVM [116] propose to protect sensitive VMs from a compromised hypervisor by extracting the security-sensitive operation of modifying the nested page tables from the hypervisor. In H-SVM, every modification of the page tables by the hypervisor is verified causing the memory pages of a sensitive VM to remain isolated. For this, the page tables are stored in a protected memory region which is only accessible for the H-SVM component, which can either be implemented as a separate controller in the processor chip or in microcode. When a physical memory page of a sensitive VM is deallocated, H-SVM is responsible for cleaning the page before it is assigned to another VM. In addition to that, H-SVM is in control of the I/O page tables used by the IOMMU to protect a sensitive VM also from malicious DMA accesses. Apart from memory isolation, H-SVM allows to authenticate sensitive VMs prior to boot and also provides a mechanism to migrate VMs between multiple H-SVM platforms. CloudVisor [264] follows the same goal. However, in contrast to H-SVM [116], the security-sensitive hypervisor functions are offloaded to an additional software layer by making use of nested virtualization. In vTZ [97], which targets the ARM platform and which we discussed in more detail in Section 2.2.1, TrustZone is used to outsource the sensitive mapping operations. HyperWall [227], another approach which provides enclaves in the form of VMs, introduces a new set of access control tables, called Confidentiality and Integrity Protection (CIP) tables, which are used to assign memory pages to enclaves. On every memory access, the MMU (or IOMMU) checks in the CIP tables if the access is valid, whereby the CIP tables are stored in a protected memory region inaccessible to system software. In a predecessor work of HyperWall [227], called Bastion [28], all security sensitive tasks are performed by the hypervisor, thus the strict separation between system software and trusted software typical for enclave architectures is not given. However, in contrast to the others works, Bastion protects VMs from physical attacks by introducing a hardware engine at the SoC which encrypts and integrity protects all VM data leaving the processor. In another predecessor work of HyperWall [227], called NoHype [119], the authors propose to drastically reduce the hypervisor size by only providing a static assignment of hardware resources which is kept fixed during VM run-time. Providing protection from physical attacks, as in Bastion [28], in combination with an untrusted hypervisor, as in CloudVisor [264], H-SVM [116] or HyperWall [227], is then achieved in HyperCoffer [257].

Inspired by the academic research on VM enclaves, AMD developed the Secure Encrypted Virtualization (SEV) [105] technology which follows a similar goal. SEV is based on the AMD Secure Memory Encryption (SME) technology and isolates VMs from each other and the untrusted hypervisor by encrypting VM code and data in memory. Thus, the high-level design of SEV is comparable to the design of HyperCoffer [257], however, SEV uses an individual encryption key for each VM. The encryption keys are created in hardware, inaccessible to system software, and are managed by the AMD Secure Processor (SP) which interacts with the hardware encryption engine to encrypt or decrypt all VM data leaving or entering the processor. When VM data is loaded into the processor cache, it is protected by an access control mechanism relying on Address Space Identifiers (ASIDs). Moreover, a VM owner can verify the initial state of his VM through remote attestation based on an identification key embedded into the firmware. In the initial release of SEV [105], the encryption mechanism only covered the VM code and data. The processor register state remained visible over the Virtual Machine Control Block (VMCB) structure which was exploited in controlled side-channel attacks [93]. In the next SEV release, called SEV-ES (Encrypted State) [106], the encryption mechanism was extended to the processor register state to prevent information leakage over the VMCB structure. SEV-ES does not guarantee integrity for the encrypted VM memory pages which was shown to be exploitable in further controlled side-channel attacks [139, 256, 170]. In the newest release of SEV, called SEV-SNP (Secure Nested Paging) [107], an additional address translation table, the Reverse Map Table (RMT), is introduced to restrict the hypervisor's access to pages belonging to VM enclaves. IBM PEF [98] also provides enclaves in the form of VMs. For this, PEF introduces a new highly privileged software component, called the Ultravisor, which consists of 39 KLOC and controls the setup of the enclaves. In contrast to SEV, PEF does not provide a hardware memory encryption engine to date. However, PEF can prevent controlled side-channel attacks on enclave page table since they are protected and verified by the trusted Ultravisor component.

Enabling confidential cloud computing by protecting sensitive VMs was only recently also addressed by Intel and ARM with new enclave architectures which are not yet publicly available in hardware. Comparable to AMD SEV or IBM PEF, Intel TDX [49] uses a hardware encryption engine, based on the Multi-Key Total-Memory Encryption (MKTME) [48] engine, to encrypt sensitive VMs in memory, whereby an unique key is used for each VM. In contrast to SEV, the encryption engine used in TDX also provides integrity protection. Moreover, the pages tables of the VMs are not stored in the memory of the untrusted hypervisor. TDX introduces a new processor mode, called Secure-Arbitration Mode (SEAM), which hosts a trusted software, the Intel TDX module. The TDX module has exclusive access to a memory region in which all the sensitive VM page tables are stored. When the boot of a VM is triggered, the hypervisor prepares the page tables and hands them over to the TDX module, which verifies that the memory regions of the sensitive VMs do not overlap before passing control to the VM. Also the recently introduced ARM CCA [152] provides enclaves in the form of VMs, called *realms*. At its core, CCA introduces a new realm processor mode which is orthogonal to the normal and secure modes introduced with ARM TrustZone. Accompanied with the new processor mode, the MMU is extended by an additional translation level which is used to search a newly introduced memory structure, called Granular Protection Table (GPT), which is stored in protected memory. With the GPT, fine-grained access rules, e.g. on a per-page basis, can be defined to isolate realms from each other and the system software. Thus, in contrast to TDX, CCA can use access control mechanisms instead of memory encryption to prevent illegal accesses to sensitive VM data. In order to protect against physical attacks, CCA-enabled systems can also include a hardware encryption and authentication engine in front of the memory controller called Memory Protection Engine (MPE).

Kernel-space Enclaves

We use the term kernel-space enclave to describe a type of enclave which also runs code, typically a runtime, in the privileged kernel space. In contrast to VM enclaves, kernel-space enclaves do not rely on management functionalities provided by a hypervisor. One early work on enclave architectures which provide kernel-space enclaves is Flicker [165] which is based on the AMD Secure Virtual Machine (SVM) technology. AMD SVM was developed to allow, together with a TPM, a late boot of security critical code, e.g. a security kernel or secure VMM. Flicker uses SVM and a TPM to verify and load a small enclave-like execution environment called Secure Loader Block (SLB) which wraps a piece of sensitive application code, called Pieces of Application Logic (PAL). To protect the SLB and PAL during run-time, Flicker suspends the untrusted OS before executing the SLB. The SLB provides a runtime for the PAL and also implements OS functionalities like memory management or peripheral handling, whereby the PAL is executed in ring 3 (user space). After the PAL execution is finished, the SLB is responsible for cleaning the processor state and memory from sensitive data before the OS gets restored. In Flicker, the TPM is used to measure the SLB code before boot for a later remote attestation. Furthermore, it can be used for a sealed storage functionality which is needed when a context switch to the OS is performed since this requires to securely store the PAL state. SICE [7] achieves a similar separation of the system in a commodity OS and kernel-space enclaves. However, instead of relying on AMD SVM, SICE uses the highly privileged System Management Mode (SMM) on Intel processors for hosting the trusted software which performs the secure context switch between the OS and an enclave. Keystone [135], developed for the open RISC-V architecture, does not suffer from Flicker's [165] main limitation that the commodity OS must be suspended while an enclave is executed. Keystone solves this problem by making use of the Physical Memory Protection (PMP) unit provided on RISC-V processors which allows to perform access control on physical memory addresses. Keystone isolates kernel-space enclaves, consisting of a sensitive application and a runtime, on physical processor cores by configuring the PMP's of all cores simultaneously so that only the core executing the enclave code has access to the memory region dedicated to the enclave. In contrast to Flicker, this allows to run enclaves in parallel with the commodity OS. The trusted software which manages the enclaves and configures the PMPs is called Security Monitor (SM) and runs in the highest-privileged software level of the processor, the machine level. Moreover, Keystone provides cache side-channel resilience for enclaves by providing a way-based partitioned LLC.

One of the first enclave architectures were based on the ARM TrustZone [151] technology. The core idea of TrustZone is to separate a system into two worlds, the *normal* world and the *secure* world. The normal world contains the commodity untrusted OS and all non-sensitive applications. The secure world comprises the sensitive applications, called Trusted Apps (TAs), which run in the secure-world user space and the Trusted OS (TOS) which runs in the secure-world privileged level. TAs can either be standalone or come bundled with a non-sensitive application running in normal-world user space. The TOS provides isolation for the TAs based on virtual address spaces. Moreover, the TOS provides all system services to the TAs. The separation of the normal and secure world is achieved in hardware by a set of security enhancements of the processor, the system bus, the cache, and additional components on the SoC like the memory controller. On

TrustZone-enabled systems, the processor can either run in normal or secure mode, depending on which world code the processor is currently executing. The processor mode is indicated by a bit flag, called the None-Secure (NS) bit, which is propagated on the system bus, e.g. when the processor is performing a memory access. The memory controller can then perform access control on the NS-bit. All SoC components that want to interact with the secure world, e.g. a fingerprint sensor, need to be NS-bit aware. The configuration of the processor NS-bit is done by a software component, called Trusted Firmware (TF), which runs on the most privileged level of an ARM processor. Because of the various functionality requirements of the TAs, the code base of the TOS deployed on commercial devices grew to a large size which offers a large attack surface for malicious or compromised TAs [27]. Thus, given our definition of enclave architectures in Section 1.1, all TrustZone-based architectures that use a TOS to manage TAs must be viewed as providing only a single kernel-space enclave controlled by the device vendor which comprises the complete secure world code, including the TF. Most of the academic proposals which are based on TrustZone, and which we describe in detail in Section 2.2.1, also provide kernel-space enclaves [265, 267, 22, 226, 143, 133, 40, 114], only vTZ [97] provides VM enclaves.

Comparison

Cure, in contrast to its related work, is the only enclave architecture which does not follow a one-size-fits-all enclave paradigm and instead provides multiple types of enclaves on one platform. As a result, Cure can support a wider range of enclave use cases and provide enclaves fitting to the requirements of sensitive applications which makes workarounds that aim to extend enclave functionality obsolete [11, 30, 5, 217, 215, 99, 86]. In the following, we focus on a comparison of Cure to enclave architectures which provide either user-space, VM, or kernel-space enclaves.

In contrast to Overshadow [35], InkTag [95], SP^3 [260], AppSec [198] and TrustzVisor [166], the user-space enclaves of Cure do not rely on virtualization technologies. One disadvantage of virtualization-based enclave architectures is that the second-level address translation introduces a permanent performance overhead. Moreover, all memory accesses from the untrusted OS must be interposed and checked which further degrades system performance. Lastly, using the privileged hypervisor layer as the trusted software component blocks its usage for hardware virtualization. In addition, when a commodity hypervisors is used, this results in a large TCB [35, 260, 95]. The user-space enclaves provided by Intel SGX [45], Iso-X [66], Sanctum [51] and MI6 [16] are protected by performing access control at the virtual-to-physical address translation in the MMU, whereas TyTAN [18] and TrustLite [126] rely on a modified MPU and TIMBER-V [253] on a combination of MPU and memory tagging hardware. Cure, in contrast, does not require invasive hardware modifications at the complex processor cores. Instead, Cure's main access control mechanism is integrated in the much simpler system bus. None of the related works, except Sanctum [51] and MI6 [16], consider controlled side-channel attacks performed from a compromised OS. Cure protects against page-table based attacks [258, 25, 245, 168, 170] by including the enclave page tables in the enclave memory and against interrupt-based attacks [236, 91, 235, 254] by allowing the enclave to register trap handlers that are called on every re-entrance of an enclave which allows the enclave to detect an abnormal interrupt behavior. The attack class of cache side-channel attacks are also only considered by Sanctum, MI6 and Cure. Sanctum and MI6 provide a page-coloring scheme which assigns cache sets to enclaves but requires to adapt the physical memory layout of the complete software stack which is impractical for commodity system software and hinders a dynamic modification of the cache resource assignment during run-time. Cure does not impose any restrictions on the physical memory layout of the system software and implements a way-based cache partitioning in which cache ways can be exclusively assigned to enclaves. In contrast to MI6, Cure does not focus on out-of-order processors and thus does not provide partitioning for microarchitectural structures which can cause minor information leaks, e.g. the MSHR. Instead, Cure proposes to prevent transient execution attacks and also side-channel attacks on core-exclusive caches by flushing all microarchitectural resources, which are shared between execution contexts, when entering or leaving an enclave.

The kernel-space enclave provided by Cure allows a comparison with another set of enclave architectures. The main difference to traditional TrustZone-based enclave architectures and approaches which rely on the same software-based isolation [265, 267], is that Cure provides an arbitrary number of enclaves instead of a single (secure world) enclave. In contrast to TEEv [143] and PrOS [133], Cure does not rely on intra-privilege-level isolation techniques which are cumbersome and require to check all enclave binaries for forbidden privileged instructions. OSP [40] and PrivateZone [114] rely on a hypervisor level for the enclave isolation leading to the disadvantages already discussed in Section 2.2.1. TrustICE [226] and Flicker [165] provide multiple enclaves and do not occupy the hypervisor level. However, in contrast to Cure, both approaches suffer from the inherent limitation that the commodity OS must be suspended while an enclave is being executed which largely limits their practicality. SICE [7] blocks the System Management Mode (SMM) of x86 processors and is limited to the memory region which hosts the SMM code. In contrast to Sanctuary [22], Cure provides side-channel resilience for its kernel-space enclaves using cache partitioning, whereas Sanctuary must completely exclude the enclave memory from the shared cache. Moreover, Sanctuary requires the secure world to manage all secure connections from enclaves to peripherals, whereas Cure allows to assign peripherals directly and exclusively to single enclaves. Cure's hardware-assisted security mechanisms at the system bus can securely connect enclaves to MMIO peripherals and DMA-capable peripherals. This enclave-to-peripheral binding is one main difference between Cure's kernel-space enclaves and the Keystone enclave architecture [135]. To defend against cache side-channel attacks, Keystone and Cure both provide way-based cache partitioning. Keystone assigns cache ways to processor cores to which enclaves are pinned, whereas Cure assigns cache ways directly to enclaves using their enclave IDs.

Cure's hardware-assisted security mechanisms also allow to extend Cure's design with VM enclaves as provided by other enclave architectures [116, 264, 227, 257, 97, 105, 49, 152]. In Cure, the VM page tables would be protected from a rogue hypervisor inside the enclave memory, aligned to the user-space enclave design. Moreover, Cure's access control mechanisms would be used to prevent a rogue hypervisor from accessing enclave memory, comparable to H-SVM [116], CloudVisor [264], vTZ [97], HyperWall [227] and ARM CCA [152], whereas Cure would not need to rely on nested-virtualization as CloudVisor [264] and would not require to check the hypervisor binaries for forbidden instructions as vTZ [97]. Using access control mechanisms marks a difference to approaches which rely on memory encryption (and authentication) engines that transparently encrypt (and integrity protect) all enclave data in memory [257, 105, 49]. In these designs, a malicious hypervisor might be able to access the enclave data but will only read junk data, whereby rogue data modifications break the data integrity and are therefore detectable. In contrast to Cure, none of the enclave architectures providing VM enclaves consider cache side-channel attacks in their design.

3.2.2 Security Research on Heterogeneous Computing Platforms

We divide the related security research that focuses on heterogeneous computing platforms into works which provide enclaves in general for platforms that comprise different computing nodes, e.g., processors, GPUs or hardware accelerators, and works which provide security mechanisms specifically for platforms that contain a Network-on-Chip (NoC) architecture.

Enclave Architectures for Heterogeneous Computing

Graviton [242] and HIX [113] focus on the Intel SGX enclave architecture and aim to extend its functionalities in a way that SGX enclaves can communicate securely with GPUs. In Gravition [242], this is achieved by customizing the GPU. The core idea of Graviton is to extend the Command Processor (CP) of the GPU by an encryption engine and new instructions so that the GPU memory management, which is performed by the GPU driver, can be verified by the CP. Moreover, cryptographic key material is provisioned to the CP which can be used to establish secure channels to multiple enclaves. This enables Graviton to share a GPU between enclaves. In Graviton, enclaves can send commands, ML algorithms and sensitive data to a GPU over their secure channel, encrypted and integrity protected. The encryption engine in the CP is only used to decrypt and verify the GPU commands, bigger data blocks are encrypted from the compute engine of the GPU to reduce the performance overhead. HIX [113] follows another approach to extend the protection of SGX to GPUs. The core idea of HIX is to provide a central enclave, called GPU enclave, which can claim exclusive access to a GPU. All customer enclaves that want to shift workloads to a GPU can establish a secure channel to the GPU over the GPU enclave. In contrast to Graviton, the authors of HIX do not change the GPU hardware. However, the authors need to change three components of a commodity SGX system. i) The authors remove (most of) the GPU driver from the OS and run it in the GPU enclave. ii) New SGX instructions and structures are added allowing the page fault handler to also check virtual addresses that target MMIO memory. iii) The Peripheral Component Interconnect Express (PCIe) root tree is slightly modified so that the routing configuration can be locked which prevents an adversary from routing data to devices under his control. All data send between the user enclave, GPU enclave and GPU is encrypted and integrity protected. As in Graviton, the compute engine of the GPU is utilized to encrypt and verify large data blocks. The GPU enclave acts as the central gatekeeper to the GPU which allows to divide the GPU between multiple user enclaves. This, however, objects the design goal of SGX to isolate all enclaves from each other. An exploitable vulnerability in the GPU enclave could be used from an adversary to compromise all enclaves relying on it.

HETEE [270] focuses on rack-scale heterogeneous computing in cloud environments and proposes a tamper-resistant chassis, called the HETEE box, which comprises microservers, called proxy nodes, and hardware accelerators, e.g., GPUs or FPGAs, connected over PCIe fabric. The proxy nodes run the necessary software stacks, e.g. ML frameworks and drivers to communicate with the accelerators and to offload computing workloads. HETEE enables a confidential computing scenario by connecting a trusted custom hardware module, called Security Controller (SC), to the PCIe fabric. When a cloud customer wants to deploy sensitive computing workloads to the cloud, the SC allocates a proxy node and hardware accelerators for the customer and configures the PCIe fabric in a way that the communication between the node and accelerators cannot be intercepted. Then, the cloud customer can send his encrypted sensitive workloads to the SC which will decrypt it and deploy it on the proxy node. After execution of the sensitive workloads, the SC sanitizes the proxy nodes and restores their initial state which requires a reboot. In contrast to Graviton [242] and HIX [113], HETEE does not provide fine-grained enclave-like compartments on the proxy nodes but always assigns a complete proxy node to a cloud customer.

Security Mechanisms for NoC-Based Platforms

Another line of security research focuses on computing systems where the heterogeneous computing nodes, e.g. processors or accelerators, are connected over an NI to an on-chip architecture consisting of a network of routers. In HERMES [122], each computing node gets a security level assigned (highly trusted, trusted, untrusted, unknown) and depending on the security level, the nodes are clustered. Sensitive and non-sensitive applications are then distributed on the nodes depending on their security level. One designated node in each cluster, called the anchor node, is responsible for the communication with other clusters of the same security level, whereas all communication is encrypted. Since only the anchor nodes perform an encrypted communication, the performance and key management overheads are reduced. In HERMES, physical memory is assumed to be distributed among all nodes. When one node sends a memory request to the physical memory of another node, it is routed through the network and then checked by the NI of the target node. Only if the source node has the correct security level to access the requested memory, the access is permitted.

Instead of encrypting the communication between the nodes to prevent malicious or compromised nodes from extracting sensitive application data, other approaches propose to use the NI as a firewall to individually control the access of every node to the network. In SEMA [164], a trusted node is introduced as a central TCB which runs a security kernel that distributes applications among the nodes and configures the NIs which contain LUTs to perform access control on physical memory addresses. Porquet et al. [186], Fernandes et al. [69] and Fiorin et al. [72] follow a similar approach by introducing table-based access control mechanisms at the NIs which allow to define permission rules in granularities ranging from single bytes to full pages.

In order to prevent the overhead induced by access control mechanisms in the NIs, Fernandes et al. [70] propose to mitigate the leakage of sensitive application data by routing data around potentially malicious nodes in the network. The authors assume a setup in which the nodes are divided into different security zones and provide a routing algorithm which prioritizes communication along paths considered secure while still guaranteeing a deadlock free packet routing. Besides protecting from a direct data access of malicious nodes, novel routing algorithms were also proposed to mitigate side-channel attacks performed on the network level, whereby Wang et al. [246] propose to always prioritize low-security traffic so that it is never affected by the demands of high-security traffic which would leak sensitive side-channel information. To prevent a denial of service, low-security traffic is bounded by a certain static bandwidth limit. In contrast, Gossip NoC [197] aims to mitigate ongoing side-channel attacks by altering routing paths when an abnormal network traffic behavior is detected through bandwidth monitoring.

Comparison

The enclave architectures Graviton [242] and HIX [113] target heterogeneous computing platforms equipped with general purpose processors and GPUs. Both approaches rely on the availability of Intel SGX on all processor cores of the system. Moreover, HIX requires modifications at the SGX hardware, whereas Graviton modifies the GPU hardware. In both approaches, all communication between enclaves and a GPU must be encrypted. Graviton and HIX are not well-suited for NoC-based architectures with a large number of diverse computing nodes since some of the computing nodes will present unmodifiable IP-blocks from third-party vendors. For these cases, it cannot be assumed that a trusted enclave architecture is implemented on the node. With a direct access to the network, these computing nodes would be able to access the sensitive enclave data distributed over the memory nodes of the system. Our proposed enclave architecture [61], which specifically targets NoC-based architectures, implements all security mechanisms in the Distributed Memory Guard (DMG) outside of the computing nodes in front of the NI. Thus, also unmodifiable computing nodes can be included in the system design without risking that a malicious or compromised node accesses the sensitive enclave data. Moreover, the DMG allows to create secure communication channels between computing nodes without relying on encrypted communication streams. HETEE [270] is focusing on rack-scale heterogeneous computing platforms comprising many different computing nodes. In contrast to the DMG-based architecture, HETEE focuses on off-chip heterogeneity where a PCIe fabric is used to connect general purpose (micro)-servers with hardware accelerators. Our DMG-based architecture focuses on on-chip heterogeneity where computing nodes are connected using a Network-on-Chip (NoC) architecture. In contrast to Graviton, HIX and the DMG-based architecture, HETEE cannot provide enclaves on a (micro)-server but only isolates complete servers and assigns hardware accelerators to them.

The main difference between our work DMG and other works which also explicitly focus on NoC-based architectures [122, 164, 186, 69, 72, 70, 246, 197] is that these proposals do not provide isolated execution environments in the form of enclaves that are protected from the system software. Instead, these works provide very coarse-grained security domains which either compromise a complete computing node or even a set of computing nodes, no matter whether the proposals' security relies on encrypted network communication streams [122], access control mechanisms in the NI [164, 186, 69, 72] comparable to the DMG, or a modified network routing behavior [70, 246, 197]. In all these cases, the security challenges arising inside of a computing node are not considered.

4

New Cache Designs enabling Side-Channel Resilient Enclaves

Since two decades, cache side-channel attacks performed from software have been one of the most active fields of security research. Over the years, researchers successfully showed, across software privilege levels, cache levels and processor architectures, that caches leak information over their microarchitecture. This information leakage can be exploited by an adversary to infer the internal state of a victim process which ultimately allows to leak sensitive data, e.g. cryptographic key material, from a victim. The root cause of these side-channel attacks is that caches, which were introduced as an optimization technique in modern processors to keep frequently used data in fast-access memory, are typically shared among all processes running on a system. The sharing of cache resources leads to a competition between all processes in which processes evict cached data from each other to store their own data in the cache. Thus, the processes influence each others run-time behavior which is what the adversary exploits to infer the internal victim state. Academic research discovered various different attack types, whereby the most investigated ones can be broadly categorized in access-based attacks [87, 261, 112], conflict-based attacks [180, 111, 155] and occupancy-based attacks [218]. In the access-based attacks, the adversary provokes cache contention by evicting the victim data from the cache using flush instructions. In the conflict-based attacks, the adversary provokes contention by filling cache lines used by the victim with its own data. In the occupancy-based attacks, the adversary observes evictions on cache lines occupied by its own data. The severity of cache side-channel attacks was also shown on enclave architectures across platforms [19, 88, 83, 53, 210, 266].

Along with the research that aims to discover new attack vectors on the cache microarchitecture, security researchers also developed a plethora of cache side-channel defenses. Apart from providing time-constant, and thus side-channel resilient, implementations of vulnerable cryptographic algorithms [46, 14], researchers proposed several software-based defenses to prevent successful attacks, e.g., by detecting on-going attacks and killing suspicious processes using performance counters [39, 184], or by preventing an adversary from performing precise time measurements [180, 163, 96, 240] which are essential for cache side-channel attacks. However, more promising are approaches which tackle the side-channel problem directly at its root cause and propose novel cache microarchitectures which are by design side-channel resilient.

In this chapter, we describe our contributions to the protection of enclaves from cache side-channel attacks using side-channel resilient cache designs (Section 4.1), summarize related research and position our work to it (Section 4.2).

4.1 Contributions

This dissertation contributes to the enclave computing research of *Side-Channel Resilient Caches* (Section 1.2.5) by designing, implementing and evaluating a novel side-channel resilient cache microarchitecture, named CHUNKED-CACHE, which is described in the following publication that can be found in Appendix E:

[62] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **CHUNKED-CACHE: On-Demand and Scalable Cache Isolation for Security Architectures**. In *Symposium on Network and Distributed System Security (NDSS)*, 2022. CORE Rank A*. Appendix E.

Microarchitectural Design. In CHUNKED-CACHE's microarchitectural design, which targets Last-Level Caches (LLCs) in systems that support enclaves, software is divided into multiple Isolated-Domains (I-Domains) and a single Non-Isolated Domain (NI-Domain), whereby each domain is identified system-wide using a Domain-ID (DID). The I-Domains represent sensitive applications encapsulated in enclaves, whereas the single NI-Domain, comprises the complete untrusted system software, including all non-sensitive applications, as depicted in Figure 6. One key aspect of CHUNKED-CACHE's design it that it provides side-channel resilience for sensitive applications (I-Domains) through fine-grained cache partitioning on a cache-set basis. Thus, in contrast to coarse-grained way-based partitioning, CHUNKED-CACHE prevents cache underutilization and scales to a larger number of protected domains. Moreover, CHUNKED-CACHE also allows to disable side-channel resilience for I-Domains if the protection is not required.

At its core, CHUNKED-CACHE introduces a new cache set indexing policy which differentiates between I-Domains and the NI-Domain when performing the memory-to-set mapping. The NI-Domain always gets a fixed set of cache sets assigned which are called the *mainstream cache* and which cannot be allocated to I-Domains, e.g., in Figure 6 the global sets 0-7 are assigned to the NI-Domain. CHUNKED-CACHE still allows the NI-Domain to allocate all unused cache sets outside of the mainstream cache with set-associative indexing. In the cache controller, this is achieved by accessing sets which map the same memory addresses as a mainstream cache set, called congruent cache sets, in parallel with the mainstream cache set (global sets 12-15 in Figure 6). For the I-Domains, the owner of a domain can decide whether the domain requires side-channel protection. If not, the I-Domain's data is cached in the mainstream cache using the same indexing policy as for the NI-Domain. However, to prevent a direct cache access from a potentially malicious or compromised NI-Domain, each cache line is tagged with the DID of the I-Domain which occupies the cache line. CHUNKED-CACHE enforces that only the owner domain can access the data, whereas the NI-Domain can still evict the I-Domain data from the

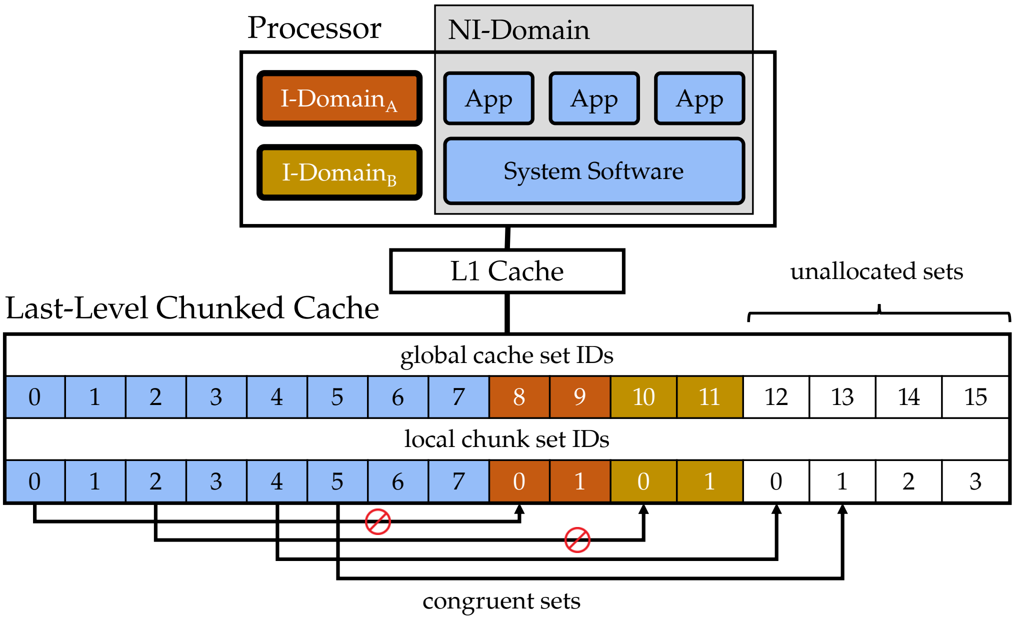


Figure 6: System view on Chunked-Cache which splits the cache sets between the NI-Domain, consisting of the system software and all non-sensitive applications, and the isolated domains I-Domain$_A$ and I-Domain$_B$.

cache which is desirable to preserve a conventional cache behavior for the NI-Domain. When an I-Domain requires cache side-channel protection, Chunked-Cache assigns a system-defined number of cache sets exclusively to the I-Domain, called the domain's *cache chunk*. In this case, no other I-Domain or the NI-Domain can access or evict cache lines from the assigned sets. Thus, Chunked-Cache's design provides a strong isolation of cache resources which also protects from occupancy-based side-channel attacks [218] since a potential victim I-Domain can also not be tricked into evicting cache lines from an adversarial domain. In Figure 6, I-Domain$_A$ gets the global cache sets 8-9 exclusively assigned and I-Domain$_B$ the sets 10-11. In the cache controller, the indexing for the I-Domains is achieved by a configurable set indexing mechanism. On every I-Domain memory access, the cache controller first verifies whether the requesting I-Domain requires side-channel protection. Then, the index bits of the memory address are used to locate the correct local set in the domain's cache chunk. Since the number of cache sets assigned to an I-Domain varies, the indexing bits must be configurable. In a second step, the local cache set is mapped to the global cache set of the LLC by querying a hardware table structure. The global cache set is then used to perform the tag comparison in the tag store and to access the correct cache line.

Apart from a strong isolation of the cache resources, Chunked-Cache allows to assign or remove cache sets to or from an I-Domain dynamically during run-time, which enables a flexible management of cache resources. In order to prevent the leakage of sensitive data, all reassigned cache sets are cleaned by the cache controller. Since Chunked-Cache's set indexing for the mainstream cache is consistent and independent from the number of cache sets occupied by I-Domains, the NI-Domain does not need to flush its own data from the cache when sets are reassigned. Thus, Chunked-Cache minimizes the performance impact on the NI-Domain. Moreover, Chunked-Cache's design also allows side-channel protected I-Domains to define cacheable memory regions which are explicitly shared with the NI-Domain, e.g. to use OS services. For these regions, Chunked-Cache guarantees inaccessibility for all other I-Domains by using cache-line tagging. Chunked-Cache can be used together with commodity replacement policies. The strict cache resource isolation in Chunked-Cache on a cache-line basis also prevents that meta-data of the replacement policy algorithm leaks between domains which was shown to be exploitable in cache side-channel attacks [23].

Implementation & Evaluation. We implement the proposed Chunked-Cache design on the Register-Transfer Level (RTL) and include our prototype into a single-core open-source RISC-V system which we synthesize for a ZedBoard Zynq-7000 FPGA to perform our hardware overhead evaluation. The logic overhead induced by the Chunked-Cache microarchitecture adds up to 1,6% of a single-issue single-core RISC-V processor. The storage overhead of our prototype accumulates to 2.3% compared to an unmodified 16 MB LLC which results in an approximate area increase of 2.7%. For Chunked-Cache's power consumption in 22 nm technology, which we evaluate using the CACTI-6.0 tool [173], we measure an overhead of 12.3% compared to an unmodified 16-way 16 MB LLC with 64 B cache line size. Our evaluation shows that the hardware overheads introduced by Chunked-Cache are reasonable. Apart from the logic, storage and power consumption overhead, we also use our hardware implementation to infer the cycle latency that Chunked-Cache incurs on memory accesses. For all accesses to the exclusively assigned cache chunks of I-Domains, an additional latency of 1 cycle is introduced, whereas for accesses to the mainstream cache 2 cycles latency are added, compared to an unmodified commodity LLC with an estimated access latency of 80 cycles. Thus, Chunked-Cache only introduces a small access latency of 1.25%-2.5%.

Besides our RTL prototype, we also implement a C++ model of Chunked-Cache which we include, together with the computed cycle latencies, into the cycle-accurate gem5 system simulator [159] in order to evaluate the performance overhead Chunked-Cache induces on software in terms of cache miss rates and the Cycles Per Instruction (CPI) metric. To evaluate Chunked-Cache in a realistic setup, we use gem5 in full-system mode and model a multi-core architecture with a 3-level cache hierarchy. Moreover, we use compute-intensive SPEC CPU2017 benchmarks [50] and I/O-intensive real-world applications for the evaluation, whereby for each experiment, we simulate at least 1 billion processor instructions. In a first set of experiments, we demonstrate the correctness of our prototype by showing the influence of Chunked-Cache's design on the cache miss rates and CPI metric when running SPEC benchmarks in I-Domains, whereby we assign a varying number of cache sets to the domains. Our results show that the cache miss rates in general heavily depend on the run-time characteristics of the specific benchmarks but decrease when more cache resources are assigned to an I-Domain, whereby the opposing behavior can be observed for the CPI metric. Additional experiments show the same behavior also for real-world applications, namely the nginx webserver [174], whereby we run the server in one I-Domain and the HTTP benchmarking tool wrk in another I-Domain and let them communicate over the network.

In another set of experiments, we evaluate Chunked-Cache's influence on the NI-Domain. We demonstrate that even when only 1/8 of the LLC cache resources are assigned to the NI-Domain, the cache miss rates decrease compared to an unmodified LLC. This is possible since Chunked-Cache enables the NI-Domain to allocate unused cache resources which we verify in an additional experiment. Moreover, we conduct further experiments to demonstrate the dynamic cache resource allocation functionality of Chunked-Cache by modifying the size of an I-Domain's cache chunk during runtime. Our evaluation shows that the performance of the NI-Domain is not negatively influenced by the cache set reassignment. The scalability of Chunked-Cache's design is verified in another experiment by running 32 I-Domains in parallel, which was the maximal number of domains our evaluation system was capable of simulating. In a last set of experiments, we compare Chunked-Cache with way-based partitioning schemes. Our experiments show the superiority of Chunked-Cache's fine-grained partitioning scheme which for an equally-sized portion of the cache assigned to an I-Domain, 1 MB and 2 MB, achieves a 43% and 39% lower cache miss rate than a way-based partitioned LLC, respectively.

4.2 Related Work

In the following, we summarize research most related to Chunked-Cache, which are works that propose new side-channel resilient cache architectures, and compare it to our work. The proposals can be categorized in two broad classes, caches which use randomization to disguise the mapping between memory addresses and caches sets, and caches which strictly partition the cache resources.

4.2.1 Randomization-based Cache Architectures

We further categorize the randomization-based approaches depending on the hardware structures used to define the randomized mappings, namely tables or functions which depend on some form of key or random seed.

Table-based Randomization

RPCache [248] was one of the first cache designs developed for mitigating software-based cache side-channel attacks. In RPCache, the authors propose to add permutation tables in front of the address decoder logic of the cache which introduce a new level of indirection in the cache by randomly selecting cache sets during cache indexing. When a cache line is replaced, RPCache selects a random cache set and swaps the indices of the originally and randomly selected cache sets in the permutation table. To prevent leaking side-channel information, all cache lines in the swapped cache sets must be invalidated. Newcache [249] follows a similar approach based on random mapping tables. However, Newcache introduces a logical direct-mapped cache whose output is mapped to the physical cache lines using the mapping tables. This allows Newcache to perform a randomized mapping on a cache line granularity which avoids the cache line invalidations necessary in RPCache's [248] design. In Newcache, the mapping tables are dynamically updated during runtime, whereby each process that requires cache side-channel protection uses its own logical random mapping table. Random Fill Cache [154] builds upon the concepts of RPCache [248] and NewCache [249] but introduces a new cache-fill strategy to prevent collision-based attacks [218] which exploit that the reuse of already cached data blocks can be observed by an adversary. In Random Fill Cache, a data block read from memory after a cache miss is not inserted into the cache but sent directly to the processor. The cache is then filled with randomized fetches from neighboring memory addresses.

Keyed Function Randomization

The main limitation of randomization schemes based on mapping tables is that they are only applicable to small core-exclusive L1 caches. On large LLCs, mapping tables cannot be efficiently deployed since the tables scale linearly which would induce an impractical hardware overhead. Unfortunately, providing side-channel resilience is especially important in LLCs since they are shared among cores and thus simply flushing sensitive data out of the cache during process context switches is not possible. Hence, more recent proposals for side-channel resilient cache designs rely on keyed functions for the randomization which allow to deploy the designs also to bigger caches. In TSCache [233], a simple hardware logic is used together with a random seed to map the tag and index bits of a memory address to a cache line, whereby each process is assigned its own seed which must be frequently reset.

In CEASER [193], an encryption function is used to perform the mapping from physical memory addresses to cache sets. The authors propose low-latency block ciphers for the encryption function which are cryptographically more secure than the functions proposed in TSCache [233]. CEASER periodically changes the key used for the encryption, so that an adversary cannot learn the secret mapping, whereby one key is used for the mappings of all processes on the system. The re-keying, which per default is performed after 100 cache accesses, requires a remapping of all data stored in the cache. In order to reduce the performance impact of the remapping, CEASER implements a gradual remapping strategy which remaps only a part of the cache at a time. The authors of CEASER select a re-keying rate of 100 cache accesses based on their analysis of state-of-the-art conflict-based cache attacks which use eviction set construction algorithms of $\mathcal{O}(E^2)$ complexity, where E is the number of cache lines in the attack pattern. In subsequent work [194], Qureshi shows in two new attacks that the complexity of an eviction set construction algorithm can be reduced to $\mathcal{O}(E)$ and that the cache replacement policy of CEASER [193] can be exploited so that re-keying rates of 35 and 1, respectively, would be required to keep CEASER's security guarantees. Thus, Qureshi proposes an improved design called Skewed-CEASER (CEASER-S) [194] which divides the cache ways into multiple partitions and provides an unique encryption key for each of the partitions. The improved design effectively provides a mapping randomization on a cache-line level which complicates the adversary's search for an eviction set. With CEASER-S, the re-keying rate can again be set to 100.

ScatterCache [255] is another approach which relies on keyed mapping functions to translate memory addresses to cache lines, whereby the authors provide a hash- and permutation-based variant of their design. Similar to CEASER-S [194], ScatterCache performs the randomization on a cache-line level. However, ScatterCache also feeds a software-managed identifier for each process (or security domain) as an input to the mapping function in order to provide independent mappings for each process.

CEASER [193], CEASER-S [194] and ScatterCache [255] follow a global randomization strategy in which each memory address can theoretically be mapped to any cache set which increases the access latency of the cache. The authors of PhantomCache [229] propose a localized randomization strategy in which each memory address can only be mapped to a subset of the cache sets using a combination of XOR operations, random salts and a hash function. The simpler randomization strategy enables a faster cache access since all sets can be checked in parallel. Unfortunately, the parallel access induces a large power consumption overhead of 67% compared to an unmodified cache.

In order to withstand even faster eviction set construction algorithms [188], Mirage [201] follows a different approach to prevent conflict-based cache attacks. Instead of randomizing the mapping from memory addresses to cache lines, Mirage randomizes the cache line selection performed during eviction to prevent conflicts that still happen within CEASER [193, 194], ScatterCache [255] and PhantomCache [229]. Mirage introduces a novel global eviction strategy in which eviction candidates are selected from all cache lines in the cache instead of just the current set, while still keeping a practical set-associative lookup strategy. As a result, Mirage removes set-associative evictions all together which are required by an adversary to build an eviction set.

4.2.2 Partitioning-based Cache Architectures

One of the earliest works which proposes partitioned caches as a side-channel defense is PLCache [248]. In PLCache, cache lines are exclusively assigned to processes through a combination of memory tagging and cache locking. To achieve this, PLCache introduces a locking bit and process identifier bits at every cache line to store the information if a cache line was locked and by which process. Only memory of this process tagged with the locking bit is allowed to evict the cache lines. The memory tagging is performed with a new set of load and store processor instructions introduced by PLCache.

In SecDCP [247], a more coarse-grained partitioning scheme is proposed in which cache ways are dynamically assigned to security classes, whereby each process is assigned a fixed security class. SecDCP adds an identifier to every cache line to indicate which security class loaded the cache line. This identifier is used to decide during cache-way reassignment, whether the complete way must be flushed or if the data can be kept in the cache. Flushing the cache is required when a cache way is reassigned to a higher security class to prevent the leakage of sensitive data. DAWG [123] is another work which proposes a way-based cache partitioning to mitigate side-channel attacks. In contrast to SecDCP [247] and PLCache [248], DAWG partitions the cache resources and the metadata used by the cache replacement policy, which can also leak sensitive information as shown in the RELOAD+REFRESH attack [23], between sensitive domains. Moreover, DAWG allows communication between sensitive domains over the shared cache.

The authors of Sharp [259] focus on inclusive cache hierarchies and introduce a new cache replacement policy to prevent contention-based cache attacks performed across processor cores. The idea of Sharp is to mitigate an adversarial eviction of victim cache lines by altering the replacement policy in a way that cache lines which are also cached in the private cache of a core (potential victim cache lines) are selected last for eviction from the shared cache. When a potential victim cache line is evicted, a per-core event counter is incremented which triggers an interrupt once a certain threshold is reached. RIC [118] also focuses on the protection of inclusive caches. However, in contrast to Sharp [259], RIC does not modify the replacement policy but the cache coherency protocol by relaxing inclusion for all process data that is private (non-shared) or read-only.

HybCache [60] combines aspects from partitioning- and randomization-based cache designs and focuses on systems in which multiple sensitive domains must be protected from each other and an untrusted non-sensitive domain. Thus, HybCache's design, in contrast to the aforementioned works, fits to the typical setup of an enclave architecture where the non-sensitive domain is represented by the untrusted OS. HybCache partitions the cache and assigns a subset of the cache ways to the sensitive domains, whereas the rest is assigned to the untrusted domain. One of the key design goals of HybCache is to reduce the performance impact of a partitioning-based cache design on the OS (untrusted domain). Thus, when the OS filled all its cache ways during run-time, it can steal cache ways originally assigned to the sensitive domains. To prevent the leakage of sensitive data, the cache ways must be flushed before reassignment. Since all sensitive domains share a set of ways, HybCache implements fully-associative cache indexing combined with a random replacement policy to prevent information leakage also between the sensitive domains. Because of the high power consumption overhead caused by the fully-associative subcache assigned to the sensitive domains, HybCache is only practical for smaller core-exclusive caches. BCE [202] also focuses on providing cache side-channel resilient enclaves. BCE enables cache partitioning in a finer granularity than all way-based partitioning approaches by assigning small groups of contiguous cache sets, called *clusters*, to enclaves which allows BCE to scale to a higher number of sensitive domains (or enclaves). The clusters assigned to an enclave can be distributed over the complete cache. Nevertheless, memory addresses are still mapped uniformly into the cluster sets. To achieve this, BCE introduces a new cache indexing hardware which uses a load-balancing hash function to uniformly spread memory addresses across the logical clusters of a domain which are then mapped to the physical clusters in the cache using table structures.

Besides the presented approaches which all require hardware modifications to implement cache partitioning, other works [156, 121, 81], typically summarized under the term *page coloring*, provide a partitioning of cache resources with commodity cache designs. The core idea of page-coloring schemes is to assign physical memory regions to sensitive domains in such a way that the cache sets to which the regions get mapped to do not overlap between the sensitive domains. Stealthmem [121] creates page coloring in a cloud scenario by intercepting all VM accesses to memory addresses which would result in the eviction of a cache line assigned to another VM. Godfrey et al. [81] also focus on a cloud scenario and implement page coloring in the Xen hypervisor [228]. CATalyst [156] builds on the Intel Cache Allocation Technology (CAT) [44] to select a subset of the cache ways for a coloring scheme which is implemented in software by only assigning memory pages to VMs that have non-conflicting cache line mappings.

4.2.3 Comparison

The research works most related to Chunked-Cache are other cache microarchitectures which achieve side-channel resilience through a partitioning of the cache resources. In contrast to PLCache [248] which requires memory tagging hardware and new load and store processor instructions, Chunked-Cache's hardware modifications are limited to the cache architecture and thus are less invasive. However, on some platforms memory tagging hardware will be added to mainstream processors thus potenially allowing to build side-channel protection mechanisms with existing hardware at low costs [150]. SecDCP [247] and DAWG [123] propose a way-based cache partitioning in which cache ways are exclusively assigned to security domains. The main limitation of way-based partitioning approaches is that the number of ways available in the cache define how many security domains can be supported. Since even on modern large shared caches the way-associativity is as low as 16 or 32, the scalability of these approaches is limited. Chunked-Cache provides a partitioning based on cache sets which scales much better since shared caches typically have multiple thousands of cache sets. Moreover, when assigning cache ways instead of cache sets, the assigned cache resources are less equally used by the security domain since the same memory address range spreads across a larger number of cache sets. Since unused cache resources can also not be accessed by other domains because of the partitioning, this leads to cache memory underutilization on the system. Further, set-based partitioning leads to less cache misses for the security domains since the associativity provided by the cache is preserved. Sharp [259] and RIC [118] modify the replacement policy and cache coherency protocol, respectively, to relax cache inclusion so that sensitive data in the cache cannot be evicted from an adversary. However, both approaches cannot defend against conflict-based side-channel attacks [180, 111, 155] and are limited to inclusive cache hierarchies.

HybCache [62], such as Chunked-Cache, focuses on enclave architectures and declares a low performance impact on the OS as one of the main design goals. The main difference between both approaches is that HybCache splits the cache only into two partitions, one partition is assigned to the OS and the other one shared by all enclaves on the system. Chunked-Cache, in contrast, assigns cache resources exclusively to single enclaves. As a result, Chunked-Cache can index the enclave cache resources set-associatively, whereas

HybCache must perform fully-associative cache indexing to prevent information leakage between the enclaves. The fully-associative indexing leads to a considerably larger power consumption. Furthermore, since the enclaves still compete over the same cache resources in HybCache, occupancy-based side-channel attacks [218] between enclaves cannot be prevented. BCE [202] is a concurrent work which also focuses on enclave architectures and assigns cache resources based on cache sets. BCE and Chunked-Cache both introduce new cache indexing hardware. The main difference between both approaches is that BCE uses a load-balancing hash function to uniformly spread memory addresses across the cache sets assigned to an enclave, whereas Chunked-Cache uses modifiable memory address index bits which are configured depending on the number of cache sets assigned to an enclave. In both cases, the mapping from the local enclave cache sets to the global cache sets are performed over table structures.

Page coloring schemes [156, 121, 81] are another set of partitioning-based approaches related to Chunked-Cache. The main limitation of theses approaches is that the coloring dictates how the code and data of an execution context must be placed in the physical memory which is especially impractical in an enclave scenario since also the untrusted potentially compromised OS must adhere to the coloring scheme. Moreover, with page coloring, the assignment of cache resources to security domains cannot be efficiently modified during run-time since this would require to alter the physical memory layout of the system. Lastly, the number of cache sets that are assigned to a security domain also automatically defines how much physical memory is available for the security domain which might not be desired. Chunked-Cache which does not impose any restrictions on the physical memory layout, allows to modify the number of cache sets assigned to enclaves during run-time, and assigns cache resources independently from the size of the physical memory region used by an enclave.

The main difference between Chunked-Cache and randomization-based side-channel resilient cache architectures [248, 249, 154, 233, 193, 194, 255, 229, 201] is that Chunked-Cache's strict partitioning of cache resources completely prevents the competition of execution contexts over the same cache resources. This competition leaves randomization-based approaches vulnerable to occupancy-based side-channel attacks [218], even though the mapping from memory addresses to cache sets and the eviction policies are randomized. Moreover, since randomization-based approaches must rely on cryptographic primitives to scale to large shared caches [233, 193, 194, 255, 229, 201], there is always a certain probability for a successful brute-force attack, whereby the probability is determined by the re-keying frequency. Unfortunately, the discovery of new faster eviction set construction algorithms can make randomization-based approaches impractical when the re-keying rate must be increased substantially [194, 188, 189, 17].

5 Conclusion and Future Research

In this chapter, we summarize the contributions of this dissertation to enclave computing research (Section 5.1) and propose future research directions (Section 5.2).

5.1 Summary

This dissertation significantly contributes to the research field of enclave computing by proposing multiple new enclave architectures across processor architectures and platforms. In the following, we summarize the contributions of this dissertation and compare them to the dissertation goals defined in Section 1.3.

Openly-accessible Enclave Computing on ARM-based Devices. We presented Sanctuary [22], an enclave architecture which reaches our goal of providing a strong hardware-assisted separation between enclaves on TrustZone-enabled platforms. Thus, in contrast to traditional TrustZone-based architectures that only provide a single enclave, Sanctuary offers an ecosystem of mutually distrusted enclaves and an unrestricted access to the protection capabilities of TrustZone without security concerns, also for third-party developers. We implemented Sanctuary on a multi-core ARM-based chip set and demonstrated its practicality by thoroughly evaluating our prototype. Moreover, we presented Offline Model Guard (OMG) [12] which is based on Sanctuary and which fulfills our goal of investigating how to extend the applicability of enclaves to sensitive Machine Learning (ML) applications. With OMG, we showcased how enclaves can be used to implement an offline keyword recognition service, using the TensorFlow lite for microcontrollers framework, which provides privacy for user data, secrecy for ML models and integrity for ML algorithms.

Flexible Enclaves Across Platforms. We presented Cure [9], a novel enclave architecture which provides multiple different *types* of enclaves on one platform including enclaves that provide execution contexts isolated on an intra-privilege level, enclaves comprising an user-space process, and self-sustained enclaves which include a runtime to perform management tasks independently from the untrusted operating system. With this multi-enclave design, Cure, in contrast to other enclave architectures, does not need to make assumptions regarding the protected sensitive applications. Depending on the requirements of the sensitive applications, the best fitting enclave type is selected. Thus, Cure reaches our goal of enabling more enclave use cases since a larger set of sensitive applications can be protected on one platform without workarounds. Moreover, Cure proposes a novel access control mechanism on the system bus with which it contributes to our goal of enabling a secure communication between enclaves and peripherals without requiring to modify the peripherals or to encrypt the communication streams. We implemented Cure for the open RISC-V architecture and thoroughly evaluated our prototype in terms of hardware and performance overhead on an evaluation setup consisting of ISA simulators, a cycle-accurate system simulator and an FPGA. Moreover, we presented Distributed Memory Guard (DMG) [61] and an enclave architecture based on the DMG with which we bring Cure's multi-enclave concept also to Network-on-Chip (NoC)-based platforms, thereby fulfilling another goal of this dissertation.

New Cache Designs enabling Side-Channel Resilient Enclaves. We presented Chunked-Cache [62], a side-channel resilient partitioning-based cache microarchitecture design which targets shared caches in enclave architectures. Chunked-Cache accomplishes our goal of a scalable cache design which assigns strongly-isolated cache resources to enclaves on a fine-grained cache-set level without inflicting a large performance degradation on the system software. We implemented Chunked-Cache in hardware and on a cycle-accurate cache simulator. We evaluated the hardware and power consumption overhead of our Chunked-Cache prototype and provided a thorough performance evaluation using standard compute-intensive benchmarks and I/O-intensive real-world applications. Moreover, we demonstrated that Chunked-Cache scales better than caches providing partitioning on a cache-way level while inducing considerably less cache misses.

5.2 Future Research

Enclave computing has been a very active field of research in the last decade. The current computing trend of application-specific computing hardware and novel commercial enclave architectures indicate that enclave computing research will continue to be an important research field in the future. In the following, we discuss how commercial enclave architectures and application-specific computing hardware might steer enclave computing research in the next decade.

The existing enclave attack research, which aims to reveal security vulnerabilities of enclave architectures, was largely driven by the introduction of Intel SGX [45] since it provided the security researchers with a commercial product, available in hardware, to verify the made security claims for enclaves through extensive security analyses. Recently, the enclave architectures Intel TDX [49] and ARM CCA [152] have been proposed which introduce new hardware-assisted security mechanisms and which claim to be resilient against many attacks to which AMD SEV [105], which follows a similar goal, has been shown vulnerable to [170, 139, 93, 141, 256]. Thus, we believe that the introduction of Intel TDX and ARM CCA in hardware will spawn a range of research works which focus on verifying whether the new enclave architectures keep what they promise. As typical in security research, the discovery of new attack vectors will motivate new defensive research that aims to prevent the newly discovered attacks or that extends the enclave architectures to circumvent design limitations, as we did with Sanctuary [22] for TrustZone-based enclave architectures. Moreover, we expect that cache designs which enable side-channel resilient enclaves, as our proposed Chunked-Cache design [62], will also be investigated in conjunction with Intel TDX and ARM CCA.

Besides research focusing on new commercial enclave architectures, we envision that future research will also aim at bringing enclaves to computing hardware apart from general purpose processors, e.g., Graphics Processing Units (GPUs), Tensor Processing Units (TPUs), or other application-specific hardware, which aligns with the general computing trend of deploying more specialized hardware as an answer to the initiating slowdown of Moore's law. On the one hand, specialized enclave architectures will bring along new challenges regarding the protection from side-channel attacks. For some hardware, the challenges will be congruent to those already faced on enclave architectures for general purpose processors and thus, existing defense mechanisms, e.g. side-channel resilient caches, might be applicable with little effort. However, other specialized hardware will introduce new microarchitectural structures with a yet unknown side-channel attack surface which will require novel defense mechanisms. On the other hand, new designs for emerging heterogeneous computing platforms will be required to enable the combination of different enclave architectures on one platform and a secure interaction between all enclaves, which aligns to the challenges of general purpose enclave architectures that we address in our works Cure [9] and Distributed Memory Guard (DMG) [61].

Bibliography

[1] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *ACM Conference on Computer and Communications Security (CCS)*, pages 743–754, 2016. ISBN 9781450341394.

[2] Adil Ahmad, Kyungtae Kim, Muhammad I. Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[3] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[4] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling Open Systems through Strong Availability Guarantees for Enclaves. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1357–1372, 2021. ISBN 9781450384544.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 689–703, 2016. ISBN 9781931971331.

[6] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[7] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. SICE: A Hardware-Level Strongly Isolated Computing Environment for X86 Multi-Core Platforms. In *ACM Conference on Computer and Communications Security (CCS)*, pages 375–388, 2011. ISBN 9781450309486.

[8] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure Multiparty Computation from SGX. In *Financial Cryptography and Data Security*, pages 477–497, 2017. ISBN 9783319709727.

[9] 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 *USENIX Security Symposium*, 2021. ISBN 9781939133243.

[10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[11] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–283, 2014. ISBN 9781931971164.

[12] 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 *Conference on Design, Automation and Test in Europe (DATE)*, pages 460–465, 2020. ISBN 9783981926347.

[13] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference (ATC)*, 2005.

[14] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The Security Impact of a New Cryptographic Library. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176, 2012. ISBN 9783642334818.

[15] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *USENIX Security Symposium*, pages 1213–1227, 2018. ISBN 9781939133045.

[16] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 42–56, 2019. ISBN 9781450369381.

[17] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1110–1123, 2020. ISBN 9781728173832.

[18] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. TyTAN: Tiny trust anchor for tiny devices. In *ACM Design Automation Conference (DAC)*, 2015.

[19] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[20] Ferdinand Brasser, Tommaso Frassetto, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, and Christian Weinert. VoiceGuard: Secure and Private Speech Processing. In *Interspeech*, pages 1303–1307, 2018.

[21] 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 *ACM Annual Computer Security Applications Conference (ACSAC)*, pages 788–800, 2019. ISBN 9781450376280.

[22] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[23] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *USENIX Security Symposium*, pages 1967–1984, 2020. ISBN 9781939133175.

[24] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization. *arXiv*, 2021. URL https://arxiv.org/pdf/2108.04575.pdf.

[25] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*, pages 1041–1056, 2017. ISBN 9781931971409.

[26] 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 *USENIX Security Symposium*, pages 991–1008, 2018. ISBN 9781939133045.

[27] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1416–1432, 2020. ISBN 9781728134970.

[28] David Champagne and Ruby B. Lee. Scalable Architectural Support for Trusted Software. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010. ISBN 9781424456598.

[29] Swarup Chandra, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. Securing Data Analytics on SGX with Randomization. In *European Symposium on Research in Computer Security (ESORICS)*, pages 352–369, 2017. ISBN 9783319664026.

[30] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (ATC)*, pages 645–658, 2017. ISBN 9781931971386.

[31] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, 2013. ISBN 9781450318709.

[32] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE Symposium on Security and Privacy (S&P)*, pages 178–194, 2018.

[33] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157, 2019. ISBN 9781728111483.

[34] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 7–18, 2017. ISBN 9781450349444.

[35] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, 2008. ISBN 9781595939586.

[36] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy (S&P)*, pages 56–71, 2016. ISBN 9781509008247.

[37] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *USENIX Security Symposium*, pages 699–716, 2021. ISBN 9781939133243.

[38] 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 *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200, 2019. ISBN 9781728111483.

[39] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real Time Detection of Cache-Based Side-Channel Attacks Using Hardware Performance Counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[40] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices. In *USENIX Annual Technical Conference (ATC)*, pages 565–578, 2016. ISBN 9781931971300.

[41] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[42] Michael Clark. rv8-bench. https://github.com/michaeljclark/rv8-bench (last visited 19/02/2022), 2021.

[43] Tobias Cloosters, Michael Rodler, and Lucas Davi. TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves. In *USENIX Security Symposium*, pages 841–858, 2020. ISBN 9781939133175.

[44] Intel Corporation. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html (last visited 19/02/2022), 2016.

[45] Intel Corporation. Intel Software Guard Extensions (Intel SGX) Developer Guide. https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-intel-sgx-developer-guide.html (last visited 19/02/2022), 2018.

[46] Intel Corporation. Intel Integrated Performance Primitives Cryptography Developer Reference. https://software.intel.com/content/www/us/en/develop/download/developer-reference-for-intel-integrated-performance-primitives-intel-ipp-cryptography.html (last visited 19/02/2022), 2019.

[47] Intel Corporation. Intel Active Management Technology Developers Guide. https://www.intel.com/content/www/us/en/develop/documentation/amt-developer-guide/top/getting-started.html (last visited 19/02/2022), 2021.

[48] Intel Corporation. Intel Architecture Memory Encryption Technologies. https://www.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf (last visited 19/02/2022), 2021.

[49] Intel Corporation. Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification. https://software.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-module-1.5-base-spec-348549001.pdf (last visited 19/02/2022), 2021.

[50] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017 (last visited 19/02/2022), 2017.

[51] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, pages 857–874, 2016. ISBN 9781931971324.

[52] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. SmashEx: Smashing SGX Enclaves Using Exceptions. In *ACM Conference on Computer and Communications Security (CCS)*, pages 779–793, 2021. ISBN 9781450384544.

[53] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. In *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, pages 171–191, 2018.

[54] 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 *USENIX Security Symposium*, pages 801–818, 2019. ISBN 9781939133069.

[55] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 191–206, 2015. ISBN 9781450328357.

[56] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: Hardware-Assisted Flow Integrity eXtension. In *ACM Design Automation Conference (DAC)*, 2015. ISBN 9781479980529.

[57] Leila Delshadtehrani, Sadullah Canakci, Boyou Zhou, Schuyler Eldridge, Ajay Joshi, and Manuel Egele. PHMon: A Programmable Hardware Monitor and Its Security Use Cases. In *USENIX Security Symposium*, pages 807–824, 2020. ISBN 9781939133175.

[58] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, and Ahmad-Reza Sadeghi. LO-FAT: Low-Overhead control Flow ATtestation in hardware. In *ACM Design Automation Conference (DAC)*, 2017. ISBN 9781450349277.

[59] 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 Security & Privacy*, 18(5):57–66, 2020.

[60] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security Symposium*, pages 451–468, 2020. ISBN 9781939133175.

[61] Ghada Dessouky, Mihailo Isakov, Michel A. Kinsy, Pouya Mahmoody, Miguel Mark, Ahmad-Reza Sadeghi, Emmanuel Stapf, and Shaza Zeitouni. Distributed Memory Guard: Enabling Secure Enclave Computing in NoC-based Architectures. In *ACM Design Automation Conference (DAC)*, pages 985–990, 2021. ISBN 9781665432740.

[62] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CHUNKED-CACHE: On-Demand and Scalable Cache Isolation for Security Architectures. In *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[63] EEMBC. CoreMark. `https://www.eembc.org/coremark/index.php` (last visited 19/02/2022), 2021.

[64] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 104–117, 2018.

[65] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[66] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 190–202, 2014. ISBN 9781479969982.

[67] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[68] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. Secure and Private Function Evaluation with Intel SGX. In *ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW)*, pages 165–181, 2019. ISBN 9781450368261.

[69] Ramon Fernandes, Bruno Oliveira, Johanna Sepúlveda, Cesar Marcon, and Fernando G. Moraes. A non-intrusive and reconfigurable access control to secure NoCs. In *IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 316–319, 2015. ISBN 9781509002467.

[70] Ramon Fernandes, César Marcon, Rodrigo Cataldo, Jarbas Silveira, Georg Sigl, and Johanna Sepúlveda. A security aware routing approach for NoC-based MPSoCs. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, 2016. ISBN 9781509027361.

[71] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software.

In *ACM Symposium on Operating System Principles (SOSP)*, pages 287–305, 2017. ISBN 9781450350853.

[72] Leandro Fiorin, Gianluca Palermo, Slobodan Lukovic, Valerio Catalano, and Cristina Silvano. Secure Memory Accesses on Networks-on-Chip. *IEEE Transactions on Computers*, 57(9):1216–1229, 2008.

[73] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. JITGuard: Hardening Just-in-time Compilers with SGX. In *ACM Conference on Computer and Communications Security (CCS)*, pages 2405–2419, 2017. ISBN 9781450349468.

[74] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *USENIX Security Symposium*, pages 83–97, 2018. ISBN 9781939133045.

[75] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and Secure Index with SGX. In *Conference on Data and Applications Security and Privacy (DBSec)*, 2017.

[76] Gal Beniamini. Full TrustZone exploit for MSM8974. http://bits-please.blogspot.com/2015/08/full-trustzone-exploit-for-msm8974.html (last visited 19/02/2022), 2015.

[77] Gal Beniamini. TrustZone Kernel Privilege Escalation (CVE-2016-2431). http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html (last visited 19/02/2022), 2015.

[78] Gal Beniamini. QSEE privilege escalation vulnerability and exploit (CVE-2015-6639). http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html (last visited 19/02/2022), 2016.

[79] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *IACR Annual Conference on Cryptographic Hardware and Embedded Systems (CHES)*, pages 251–261, 2001. ISBN 9783540447092.

[80] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *ACM Symposium on Operating System Principles (SOSP)*, pages 193–206, 2003. ISBN 1581137575.

[81] Michael Godfrey and Mohammad Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE Transactions on Cloud Computing*, 2(4):395–408, 2014.

[82] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix Freiling, and Ingrid Verbauwhede. Soteria: Offline software protection within low-cost embedded devices. In *ACM Annual Computer Security Applications Conference (ACSAC)*, pages 241–250, 2015. ISBN 9781450336826.

[83] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *ACM European Workshop on Systems Security (EuroSec)*, 2017. ISBN 9781450349352.

[84] Trusted Computing Group. Trusted Platform Module Library Part 1: Architecture. https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf (last visited 19/02/2022), 2019.

[85] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium*, pages 217–233, 2017. ISBN 9781931971409.

[86] 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 *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 488–501, 2017.

[87] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games – bringing access-based cache attacks on aes to practice. In *IEEE Symposium on Security and Privacy (S&P)*, pages 490–505, 2011. ISBN 9780769544021.

[88] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX Annual Technical Conference (ATC)*, pages 299–312, 2017. ISBN 9781931971386.

[89] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.

[90] 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 *Conference on Computer Vision and Pattern Recognition*, pages 3300–3309, 2021.

[91] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. SGXlinger: A New Side-Channel Attack Vector Based on Interrupt Latency Against Enclave Execution. In *IEEE International Conference on Computer Design (ICCD)*, pages 108–114, 2018. ISBN 9781538684771.

[92] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *ACM Asia-Pacific Workshop on Workshop on Systems (APSys)*, pages 19–24, 2010. ISBN 9781450301954.

[93] Felicitas Hetzelt and Robert Buhren. Security Analysis of Encrypted Virtual Machines. *ACM SIGPLAN Notices*, 52(7):129–142, 2017.

[94] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *ACM Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013. ISBN 9781450321181.

[95] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278, 2013. ISBN 9781450318709.

[96] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Symposium on Security and Privacy (S&P)*, pages 8–20, 1991. ISBN 0818621680.

[97] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vTZ: Virtualizing ARM TrustZone. In *USENIX Security Symposium*, pages 541–556, 2017. ISBN 9781931971409.

[98] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. Confidential Computing for OpenPOWER. In *ACM European Conference on Computer Systems (EuroSys)*, 2021. ISBN 9781450383349.

[99] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 533–549, 2016. ISBN 9781931971331.

[100] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving Machine Learning as a Service. *arXiv*, 2018. URL https://arxiv.org/pdf/1803.05961.pdf.

[101] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure Computing with Cloud GPUs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 817–833, 2020. ISBN 9781939133137.

[102] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level directional predictor based side-channel attack against sgx. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, pages 321–347, 2020.

[103] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *IEEE Consumer Communications and Networking Conference*, pages 257–261, 2008. ISBN 9781450328357.

[104] Nick Hynes, Raymond Cheng, and Dawn Song. Efficient Deep Learning on Multi-Source Private Data. *arXiv*, 2018. URL https://arxiv.org/pdf/1807.06689.pdf.

[105] Advanced Micro Devices Inc. AMD Memory Encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf (last visited 19/02/2022), 2016.

[106] Advanced Micro Devices Inc. Protecting VM Register State with SEV-ES. https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf (last visited 19/02/2022), 2017.

[107] Advanced Micro Devices Inc. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf (last visited 19/02/2022), 2020.

[108] Apple Inc. About the Apple T2 security chip. https://support.apple.com/en-us/HT208862 (last visited 19/02/2022), 2020.

[109] FIRST.Org Inc. Common Vulnerability Scoring System version 3.1. https://www.first.org/cvss/v3-1/cvss-v31-specification_r1.pdf (last visited 19/02/2022).

[110] Qualcomm Technologies Inc. Qualcomm Secure Processing Unit (SPU). https://csrc.nist.gov/CSRC/media/projects/cryptographic-module-validation-program/documents/security-policies/140sp3549.pdf (last visited 19/02/2022), 2019.

[111] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security and Privacy (S&P)*, pages 591–604, 2015. ISBN 9781450342339.

[112] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 35–364, 2016. ISBN 9781450342339.

[113] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 455–468, 2019. ISBN 9781450362405.

[114] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. PrivateZone: Providing a Private Execution Environment Using ARM TrustZone. *IEEE Transactions on Dependable and Secure Computing*, 15(5):797–810, 2018.

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

[116] Seongwook Jin, Jeongseob Ahn, Sanghoon Cha, and Jaehyuk Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 272–283, 2011. ISBN 9781450310536.

[117] Scott Johnson, Dominic Rizzo, Parthasarathy Ranganathan, Jon McCune, and Richard Ho. Titan: enabling a transparent silicon root of trust for Cloud. In *Hot Chips: A Symposium on High Performance Chips*, volume 194, 2018.

[118] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed Inclusion Caches for Mitigating LLC Side-Channel Attacks. In *ACM Design Automation Conference (DAC)*, 2017. ISBN 9781450349277.

[119] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 350–361, 2010. ISBN 9781450300537.

[120] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. VoLTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium*, pages 1445–1461, 2020. ISBN 9781939133175.

[121] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*, pages 189–204, 2012. ISBN 978931971959.

[122] Michel A. Kinsy, Shreeya Khadka, Mihailo Isakov, and Anam Farrukh. Hermes: Secure heterogeneous multicore architecture design. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 14–20, 2017. ISBN 9781538639290.

[123] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987, 2018.

[124] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating System Principles (SOSP)*, pages 207–220, 2009. ISBN 9781605587523.

[125] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *IACR Annual International Cryptology Conference (CRYPTO)*, pages 104–113, 1996. ISBN 9783540686972.

[126] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *ACM European Conference on Computer Systems (EuroSys)*, 2014. ISBN 9781450327046.

[127] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[128] Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. On-Board Credentials with Open Provisioning. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 104–115, 2009. ISBN 9781605583945.

[129] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy Enhanced Secure Object Store. In *ACM European Conference on Computer Systems (EuroSys)*, 2018. ISBN 9781450355841.

[130] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems (NIPS)*, pages 1097–1105, 2012.

[131] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N. Asokan, Andrew Simpson, and Robin Ankele. Exploring the Use of Intel SGX for Secure Many-Party Applications. In *ACM Workshop System Software for Trusted Execution (SysTEX)*, 2016. ISBN 9781450346702.

[132] KVM. Documents — KVM. https://www.linux-kvm.org/index.php?title=Documents&oldid=173827 (last visited 19/02/2022), 2017.

[133] Donghyun Kwon, Jiwon Seo, Yeongpil Cho, Byoungyoung Lee, and Yunheung Paek. PrOS: Light-Weight Privatized Secure OSes in ARM TrustZone. *IEEE Transactions on Mobile Computing*, 19(6):1434–1447, 2020.

[134] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. Sanctorum: A lightweight security monitor for secure enclaves. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 1142–1147, 2019. ISBN 9783981926323.

[135] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *ACM European Conference on Computer Systems (EuroSys)*, 2020. ISBN 9781450368827.

[136] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *USENIX Security Symposium*, pages 523–539, 2017. ISBN 9781931971409.

[137] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, pages 557–574, 2017. ISBN 9781931971409.

[138] 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 *ACM International Conference on Mobile Computing and Networking (MOBICOM)*, 2019. ISBN 9781450361699.

[139] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security Symposium*, pages 1257–1272, 2019. ISBN 9781939133069.

[140] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CrossLine: Breaking "Security-by-Crash" Based Memory Isolation in AMD SEV. In *ACM Conference on Computer and Communications Security (CCS)*, pages 293–2950, 2021. ISBN 9781450384544.

[141] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security Symposium*, pages 717–732, 2021. ISBN 9781939133243.

[142] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1541–1541, 2022.

[143] Wenhao Li, Yubin Xia, Long Lu, Haibo Chen, and Binyu Zang. TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 2–16, 2019. ISBN 9781450360203.

[144] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A Two-Way Sandbox for x86 Native Code. In *USENIX Annual Technical Conference (ATC)*, pages 409–420, 2014. ISBN 9781931971102.

[145] J. Liedtke. On Micro-Kernel Construction. In *ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, 1995. ISBN 0897917154.

[146] 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 *USENIX Security Symposium*, pages 177–194, 2019. ISBN 9781939133069.

[147] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. PACStack: an Authenticated Call Stack. In *USENIX Security Symposium*, pages 357–374, 2021. ISBN 9781939133243.

[148] ARM Limited. ARM CoreLink TZC-400 TrustZone Address Space Controller Technical Reference Manual. https://documentation-service.arm.com/static/5e907a968259fe2368e2b196?token= (last visited 19/02/2022), 2014.

[149] ARM Limited. TrustZone technology for the ARMv8-M architecture. https://documentation-service.arm.com/static/5f872f95405d955c5176deff (last visited 19/02/2022), 2017.

[150] ARM Limited. Armv8.5-A Memory Tagging Extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf (last visited 19/02/2022), 2019.

[151] ARM Limited. TrustZone for Armv8-A. https://documentation-service.arm.com/static/602167b6873dd96c4deaf49b (last visited 19/02/2022), 2020.

[152] ARM Limited. Introducing Arm Confidential Compute Architecture. https://documentation-service.arm.com/static/61238c85d5c3af0155491c4f (last visited 19/02/2022), 2021.

[153] Linaro Limited. OP-TEE Trusted OS. https://github.com/OP-TEE/optee_os (last visited 19/02/2022), 2021.

[154] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 203–215, 2014. ISBN 9781479969982.

[155] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, pages 605–622, 2015. ISBN 9781467369497.

[156] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 406–418, 2016.

[157] Google LLC. TensorFlow Lite for Mikrocontrollers. https://www.tensorflow.org/lite/microcontrollers (last visited 19/02/2022), 2021.

[158] Google LLC. Zircon. https://fuchsia.dev/fuchsia-src/concepts/kernel (last visited 19/02/2022), 2021.

[159] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, et al. The gem5 simulator: Version 20.0+. *arXiv*, 2020. URL https://arxiv.org/pdf/2007.03152.pdf.

[160] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[161] Tarjei Mandt, Mathew Solnik, and David Wang. Demystifying the Secure Enclave Processor. In *Black Hat Briefings USA*, 2016.

[162] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2007. ISBN 9780387308579.

[163] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 118–129, 2012. ISBN 9781467304764.

[164] Ramya Jayaram Masti, Devendra Rai, Claudio Marforio, and Srdjan Capkun. Isolated Execution on Many-core Architectures. *IACR Cryptology ePrint Archive*, 2014. URL https://eprint.iacr.org/2014/136.pdf.

[165] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. *ACM SIGOPS Operating Systems Review*, 42(4):315–328, 2008.

[166] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 143–158, 2010. ISBN 9781424468959.

[167] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *ACM Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013. ISBN 9781450321181.

[168] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *USENIX Security Symposium*, pages 469–486, 2020. ISBN 9781939133175.

[169] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: Toward Snoop-Based Kernel Integrity Monitor. In *ACM Conference on Computer and Communications Security (CCS)*, pages 28–37, 2012. ISBN 9781450316514.

[170] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *ACM European Workshop on Systems Security (EuroSec)*, 2018. ISBN 9781450356527.

[171] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1466–1482, 2020. ISBN 9781728134970.

[172] Pascal Nasahl, Robert Schilling, Mario Werner, and Stefan Mangard. HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 187–199, 2021. ISBN 9781450382878.

[173] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A Tool to Model Large Caches. `https://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf` (last visited 19/02/2022), 2009.

[174] Clément Nedelcu. *Nginx HTTP Server*. Packt Publishing, 2013.

[175] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *USENIX Security Symposium*, pages 479–498, 2013. ISBN 9781931971034.

[176] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, and Ahmad-Reza Sadeghi. HardScope: Hardening Embedded Systems Against Data-Oriented Attacks. In *ACM Design Automation Conference (DAC)*, 2019. ISBN 9781450367257.

[177] University of California. Spike RISC-V ISA Simulator. `https://github.com/riscv-software-src/riscv-isa-sim` (last visited 19/02/2022), 2021.

[178] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*, pages 619–636, 2016. ISBN 9781931971324.

[179] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *USENIX Annual Technical Conference (ATC)*, pages 227–240, 2018. ISBN 9781939133014.

[180] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. *IACR Cryptology ePrint Archive*, 2005. URL `https://eprint.iacr.org/2005/271.pdf`.

[181] Serkan Özkan. CVE Details. `https://www.cvedetails.com/` (last visited 19/02/2022).

[182] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying Cloud Benchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–132, 2016.

[183] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested Enclave: Supporting Fine-Grained Hierarchical Isolation with SGX. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 776–789, 2020. ISBN 9781728146614.

[184] Mathias Payer. HexPADS: a platform to detect stealth attacks. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 138–154, 2016. ISBN 9783319308067.

[185] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *USENIX Security Symposium*, pages 1039–1056, 2020. ISBN 9781939133175.

[186] Joël Porquet, Alain Greiner, and Christian Schwarz. NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs. In *Conference on Design, Automation and Test in Europe (DATE)*, 2011. ISBN 9783981080186.

[187] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database Using SGX. In *IEEE Symposium on Security and Privacy (S&P)*, pages 264–278, 2018. ISBN 9781538643532.

[188] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. *arXiv*, 2019. URL https://arxiv.org/pdf/1908.03383.pdf.

[189] Antoon Purnal, Lukas Gainer, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In *IEEE Symposium on Security and Privacy (S&P)*, pages 987–1002, 2021.

[190] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *IEEE Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, 2019. ISBN 9781728135441.

[191] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies. In *ACM Conference on Computer and Communications Security (CCS)*, pages 195–209, 2019. ISBN 9781450367479.

[192] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. SecureTF: A Secure TensorFlow Framework. In *ACM International Middleware Conference*, pages 44–59, 2020. ISBN 9781450381536.

[193] Moinuddin K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 775–787, 2018. ISBN 9781538662403.

[194] Moinuddin K. Qureshi. New Attacks and Defense for Encrypted-Address Cache. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 360–371, 2019.

[195] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1852–1867, 2021. ISBN 9781728189345.

[196] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-Only Implementation of a TPM Chip. In *USENIX Security Symposium*, pages 841–856, 2016. ISBN 9781931971324.

[197] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, and Johanna Sepúlveda. Gossip NoC – Avoiding Timing Side-Channel Attacks through Traffic Management. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 601–606, 2016. ISBN 9781467390392.

[198] Jianbao Ren, Yong Qi, Yuehua Dai, Xiaoguang Wang, and Yi Shi. AppSec: A Safe Execution Environment for Security Sensitive Applications. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 187–199, 2015. ISBN 9781450334501.

[199] Nick Roessler and André DeHon. Protecting the Stack with Metadata Policies and Tagged Hardware. In *IEEE Symposium on Security and Privacy (S&P)*, pages 478–495, 2018. ISBN 9781538643532.

[200] Dan Rosenberg. Reflections on Trusting TrustZone. In *Black Hat Briefings USA*, 2014.

[201] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX Security Symposium*, pages 1379–1396, 2021. ISBN 9781939133243.

[202] Gururaj Saileshwar, Sanjay Kariyappa, and Moinuddin Qureshi. Bespoke Cache Enclaves: Fine-Grained and Scalable Isolation from Cache Side-Channels via Flexible Set-Partitioning. In *IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.

[203] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 57–68, 2011. ISBN 9781450304726.

[204] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 67–80, 2014. ISBN 9781450323055.

[205] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[206] Alexander Schlögl and Rainer Böhme. ENNclave: Offline Inference with Model Confidentiality. In *ACM Workshop on Artificial Intelligence and Security (AISec)*, pages 93–104, 2020. ISBN 9781450380942.

[207] Martin Schönstedt, Ferdinand Brasser, Patrick Jauernig, Emmanuel Stapf, and Ahmad-Reza Sadeghi. SafeTEE: Combining Safety and Security on ARM-based Microcontrollers. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 520–525, 2022. ISBN 9783981926361.

[208] 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. In *USENIX Security Symposium*, pages 1677–1694, 2020. ISBN 9781939133175.

[209] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy (S&P)*, pages 38–54, 2015.

[210] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 3–24, 2017. ISBN 9783319608761.

[211] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM Conference on Computer and Communications Security (CCS)*, pages 753–768, 2019. ISBN 9781450367479.

[212] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[213] Di Shen. Attacking Your Trusted Code: Exploiting TrustZone on Android. In *Black Hat Briefings USA*, 2015.

[214] Di Shen. Defeating Samsung KNOX with Zero Privilege. In *Black Hat Briefings USA*, 2017.

[215] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 955–970, 2020. ISBN 9781450371025.

[216] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[217] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[218] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*, pages 639–656, 2019. ISBN 9781939133069.

[219] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying Confidentiality of Enclave Programs. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1169–1184, 2015. ISBN 9781450338325.

[220] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A Design and Verification Methodology for Secure Isolated Regions. *ACM SIGPLAN Notices*, 51(6):665–681, 2016.

[221] Wilson Snyder. VERILATOR. https://veripool.org/ftp/verilator_doc.pdf (last visited 19/02/2022), 2021.

[222] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1–17, 2016. ISBN 9781509008247.

[223] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *ACM European Conference on Computer Systems (EuroSys)*, pages 209–222, 2010. ISBN 9781605585772.

[224] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks*, pages 344–361, 2010. ISBN 9783642161612.

[225] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *ACM International Conference on Supercomputing (ICS)*, pages 160–171, 2003. ISBN 1581137338.

[226] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 367–378, 2015. ISBN 9781479969982.

[227] Jakub Szefer and Ruby B. Lee. Architectural Support for Hypervisor-Secure Virtualization. *ACM SIGPLAN Notices*, 47:437–450, 2012.

[228] Chris Takemura and Luke S. Crawford. *The book of Xen: a practical guide for the system administrator*. No Starch Press, 2010.

[229] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[230] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, pages 1057–1074, 2017. ISBN 9781931971409.

[231] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and Secure DNN Inference. *arXiv*, 2018. URL https://arxiv.org/pdf/1810.00602.pdf.

[232] Florian Tramèr and Dan Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *International Conference on Learning Representations (ICLR)*, 2019.

[233] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-Channel Attacks and Time-Predictability in High-Performance Critical Real-Time Systems. In *ACM Design Automation Conference (DAC)*, 2018. ISBN 9781538641149.

[234] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security Symposium*, pages 1221–1238, 2019. ISBN 9781939133069.

[235] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *ACM Workshop System Software for Trusted Execution (SysTEX)*, 2017. ISBN 9781450350976.

[236] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *ACM Conference on Computer and Communications Security (CCS)*, pages 178–195, 2018. ISBN 9781450356930.

[237] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave

Shielding Runtimes. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1741–1758, 2019. ISBN 9781450367479.

[238] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-Flight Data Load. In *IEEE Symposium on Security and Privacy (S&P)*, pages 88–105, 2019. ISBN 9781538666609.

[239] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *IEEE Symposium on Security and Privacy (S&P)*, pages 339–354, 2021. ISBN 9781728189345.

[240] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *ACM SIGSAC Conference on Cloud Computing Security Workshop (CCSW)*, pages 41–46, 2011. ISBN 9781450310048.

[241] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with Code-centric memory Domains. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 469–480, 2014. ISBN 9781479943944.

[242] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 681–696, 2018. ISBN 9781939133083.

[243] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. RusTEE: Developing Memory-Safe ARM TrustZone Applications. In *ACM Annual Computer Security Applications Conference (ACSAC)*, pages 442–453, 2020. ISBN 9781450388580.

[244] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. RT-TEE: Real-time System Availability for Cyber-physical Systems using ARM TrustZone. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1573–1573, 2022.

[245] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *ACM Conference on Computer and Communications Security (CCS)*, pages 2421–2434, 2017. ISBN 9781450349468.

[246] Yao Wang and G. Edward Suh. Efficient Timing Channel Protection for On-Chip Networks. In *IEEE/ACM Symposium on Networks-on-Chip (NOCS)*, pages 142–151, 2012. ISBN 9781467309738.

[247] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *ACM Design Automation Conference (DAC)*, 2016.

[248] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 494–505, 2007. ISBN 9781595937063.

[249] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 83–93, 2008. ISBN 9781424428366.

[250] Pete Warden. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv*, 2018. URL https://arxiv.org/pdf/1804.03209.pdf.

[251] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy (S&P)*, pages 20–37, 2015. ISBN 9781467369497.

[252] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Async-Shock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *European Symposium on Research in Computer Security (ESORICS)*, pages 440–457, 2016. ISBN 9783319457444.

[253] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[254] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 73–85, 2019. ISBN 9781450367523.

[255] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, pages 675–692, 2019. ISBN 9781939133069.

[256] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1483–1496, 2020. ISBN 9781728134970.

[257] Yubin Xia, Yutao Liu, and Haibo Chen. Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 246–257, 2013. ISBN 9781467355872.

[258] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P)*, pages 640–656, 2015. ISBN 9781467369497.

[259] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Atacks. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 347–360, 2017.

[260] Jisoo Yang and Kang G. Shin. Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2008. ISBN 9781595937964.

[261] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, pages 719–732, 2014. ISBN 9781931971157.

[262] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (S&P)*, pages 79–93, 2009. ISBN 9780769536330.

[263] Google Project Zero. Trust Issues: Exploiting TrustZone TEEs. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html (last visited 19/02/2022), 2017.

[264] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *ACM Symposium on Operating System Principles (SOSP)*, pages 203–216, 2011. ISBN 9781450309776.

[265] Ning Zhang, Kun Sun, Wenjing Lou, and Y. Thomas Hou. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *IEEE Symposium on Security and Privacy (S&P)*, pages 72–90, 2016. ISBN 9781509008247.

[266] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive*, 2016. URL https://eprint.iacr.org/2016/980.pdf.

[267] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. SecTEE: A Software-Based Approach to Secure Enclave Architecture Using TEE. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1723–1740, 2019. ISBN 9781450367479.

[268] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 283–298, 2017. ISBN 9781931971379.

[269] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-Based Fault Isolation for ARM. In *ACM Conference on Computer and Communications Security (CCS)*, pages 558–569, 2014. ISBN 9781450329576.

[270] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, and Dan Meng. Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1450–1465, 2020. ISBN 9781728134970.

List of Figures

List of Acronyms

ASID	Address Space Identifier
CCA	Confidential Compute Architecture
DMA	Direct Memory Access
DMG	Distributed Memory Guard
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DRM	Digital Rights Management
FPGA	Field Programmable Gate Array
GIC	General Interrupt Controller
GPU	Graphics Processing Unit
IOMMU	Input/Output Memory Management Unit
IoT	Internet of Things
IPI	Inter-Processor Interrupt
ISA	Instruction Set Architecture
LLC	Last-Level Cache
LUT	Lookup Table
ML	Machine Learning
MLaas	Machine-Learning-as-a-service
MMIO	Memory Mapped Input/Output
MMU	Memory Management Unit
MPU	Memory Protection Unit
MSHR	Miss Status Handling Registers
NI	Network Interface
NoC	Network-on-Chip
OMG	Offline Model Guard
OS	Operating System
PCIe	Peripheral Component Interconnect Express
PEF	Protected Execution Facility

PoP	Package-on-Package
PTW	Page Table Walker
RTL	Register-Transfer Level
RTM	Root of Trust for Measurement
SDK	Software Development Kit
SEV	Secure Encrypted Virtualization
SGX	Software Guard Extensions
SMM	System Management Mode
SoC	System-on-Chip
TCB	Trusted Computing Base
TDX	Trust Domain Extensions
TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
TPM	Trusted Platform Module
VM	Virtual Machine
VMCB	Virtual Machine Control Block
VMM	Virtual Machine Monitor

List of Publications

Peer-reviewed Publications

Martin Schönstedt, Ferdinand Brasser, Patrick Jauernig, Emmanuel Stapf, and Ahmad-Reza Sadeghi. **SafeTEE: Combining Safety and Security on ARM-based Microcontrollers**. In *Conference on Design, Automation and Test in Europe (DATE)*, pages 520-525, 2022. CORE Rank B.

Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **CHUNKED-CACHE: On-Demand and Scalable Cache Isolation for Security Architectures**. In *Symposium on Network and Distributed System Security (NDSS)*, 2022. CORE Rank A*. Appendix E.

Ghada Dessouky, Mihailo Isakov, Michel A. Kinsy, Pouya Mahmoody, Miguel Mark, Ahmad-Reza Sadeghi, Emmanuel Stapf, and Shaza Zeitouni. **Distributed Memory Guard:**

Enabling Secure Enclave Computing in NoC-based Architectures. In *ACM Design Automation Conference (DAC)*, pages 985-990, 2021. CORE Rank A. Appendix D.

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 *USENIX Security Symposium*, 2021. CORE Rank A*. Appendix C.

Ghada Dessouky, Emmanuel Stapf, and Ahmad-Reza Sadeghi. **Enclave Computing on RISC-V: A Brighter Future for Security?**. In *Workshop on Secure RISC-V Architecture Design (SECRISC-V)*, 2020.

Shaza Zeitouni, Emmanuel Stapf, Hossein Fereidooni, and Ahmad-Reza Sadeghi. **On the Security of Strong Memristor-based Physically Unclonable Functions**. In *ACM Design Automation Conference (DAC)*, 2020. 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 *Conference on Design, Automation and Test in Europe (DATE)*, pages 460-465, 2020. CORE Rank B. Appendix B.

Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **SANCTUARY: ARMing TrustZone with User-space Enclaves**. In *Symposium on Network and Distributed System Security (NDSS)*, 2019. CORE Rank A*. Appendix A.

Magazine Articles

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**. In *Foundations and Trends in Privacy and Security, vol. 3, no. 2, pp 103-213*, 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**. In *IEEE Security & Privacy, vol. 18, no. 5, pp. 57-66*, 2020.

Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **Trusted Execution Environments: Properties, Applications, and Challenges**. In *IEEE Security & Privacy, vol. 18, no. 2, pp. 56-60*, 2020.

Invited Papers

Emmanuel Stapf, Patrick Jauernig, Ferdinand Brasser, and Ahmad-Reza Sadeghi. **In Hardware We Trust? From TPM to Enclave Computing on RISC-V**. In *IFIP/IEEE International Conference on Very Large Scale Integration*, 2021.

Ghada Dessouky, Patrick Jauernig, Nele Mertens, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **INVITED: AI Utopia or Dystopia - On Securing AI Platforms**. In *ACM Design Automation Conference (DAC)*, 2020.

Huili Chen, Siam Umar Hussain, Fabian Boemer, Emmanuel Stapf, Ahmad-Reza Sadeghi, Farinaz Koushanfar, and Rosario Cammarota. **INVITED: Developing Privacy-preserving AI Systems: The Lessons learned**. In *ACM Design Automation Conference (DAC)*, 2020.

Lejla Batina, Patric Jauernig, Nele Mertens, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **INVITED: In Hardware We Trust: Gains and Pains of Hardware-assisted Security**. In *ACM Design Automation Conference (DAC)*, 2019.

Posters

Sebastian P. Bayerl, Ferdinand Brasser, Christoph Busch, Tommaso Frassetto, Patrick Jauernig, Jascha Kolberg, Andreas Nautsch, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, Amos Treiber, and Christian Weinert. **POSTER: Privacy-preserving speech processing via STPC and TEEs**. In *ACM CCS Workshop on Privacy Preserving Machine Learning (PPML)*, 2019.

Whitepapers & Technical Reports

Ferdinand Brasser, Patrick Jauernig, Frederik Pustelnik, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **Trusted Container Extensions for Container-based Confidential Computing**. In *arXiv:2205.05747*, 2022.

Rosario Cammarota, Matthias Schunter, Anand Rajan, Fabian Boemer, Ágnes Kiss, Amos Treiber, Christian Weinert, Thomas Schneider, Emmanuel Stapf, Ahmad-Reza Sadeghi, Daniel Demmler, Huili Chen, Siam Umar Hussain, Sadegh Riazi, Farinaz Koushanfar, Saransh Gupta, Tajan Simunic Rosing, Kamalika Chaudhuri, Hamid Nejatollahi, Nikil Dutt, Mohsen Imani, Kim Laine, Anuj Dubey, Aydin Aysu, Fateme Sadat Hosseini, Chengmo Yang, Eric Wallace, and Pamela Norton. **Trustworthy AI Inference Systems: An Industry Research View**. In *arXiv:2008.04449*, 2020.

Appendices

A

SANCTUARY: ARMing TrustZone with User-space Enclaves (NDSS'19)

[22] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **SANCTUARY: ARMing TrustZone with User-space Enclaves**. In *Symposium on Network and Distributed System Security (NDSS)*, 2019. CORE Rank A*. Appendix A.

SANCTUARY: ARMing TrustZone with User-space Enclaves

Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, Emmanuel Stapf
Technische Universität Darmstadt, Germany
{ferdinand.brasser, david.gens, patrick.jauernig, ahmad.sadeghi, emmanuel.stapf}@trust.tu-darmstadt.de

Abstract—**ARM TrustZone is one of the most widely deployed security architecture providing Trusted Execution Environments (TEEs). Unfortunately, its usage and potential benefits for application developers and end users are largely limited due to restricted deployment policies imposed by device vendors. Restriction is enforced since every Trusted App (TA) increases the TEE's attack surface: any vulnerable or malicious TA can compromise the system's security. Hence, deploying a TA requires mutual trust between device vendor and application developer, incurring high costs for both. Vendors work around this by offering interfaces to selected TEE functionalities, however, these are not sufficient to securely implement advanced mobile services like banking. Extensive discussion of Intel's SGX technology in academia and industry has unveiled the demand for an unrestricted use of TEEs, yet no comparable security architecture for mobile devices exists to this day.**

We propose SANCTUARY, the first security architecture which allows unconstrained use of TEEs in the TrustZone ecosystem without relying on virtualization. SANCTUARY enables execution of security-sensitive apps within strongly isolated compartments in TrustZone's *normal world* comparable to SGX's user-space enclaves. In particular, we leverage TrustZone's versatile Address-Space Controller available in current ARM System-on-Chip reference designs, to enforce two-way hardware-level isolation: (i) security-sensitive apps are shielded against a compromised normal-world OS, while (ii) the system is also protected from potentially malicious apps in isolated compartments. Moreover, moving security-sensitive apps from the TrustZone's *secure world* to isolated compartments minimizes the TEE's attack surface. Thus, mutual trust relationships between device vendors and developers become obsolete: the full potential of TEEs can be leveraged.

We demonstrate practicality and real-world benefits of SANCTUARY by thoroughly evaluating our prototype on a HiKey 960 development board with microbenchmarks and a use case for one-time password generation in two-factor authentication.

I. INTRODUCTION

Mobile devices have already changed our daily lives in various ways. Their success can mainly be attributed to the ecosystem that evolved around them. The increasing computing and storage capabilities, the vast number and variety of apps available on app stores and markets, as well as the connectivity to cloud services make mobile devices convenient replacements for traditional computing platforms, and the de-facto standard way of accessing the Internet [40].

Despite all benefits, today's mobile devices provide a large attack surface imposing many security and privacy challenges on their system design to be able to protect sensitive applications such as mobile banking, payments, and eID services.

The TrustZone security architecture was motivated mainly by the need for *secure mobile services* [5], when introduced in 2008 as part of an industry effort. TrustZone introduces the notion of a *normal world* and a *secure world*. While the normal world runs the Legacy OS (LOS) and user-level applications, security-sensitive applications can be executed (partially or entirely) within the secure world which represents a Trusted Execution Environment (TEE) on top of the TrustZone kernel and hardware.

Problems of TrustZone. Despite TrustZone's implementation and wide-spread deployment, TrustZone-based TEEs are mainly used by the vendors for own purposes, and hence a flourishing landscape of secure mobile services is largely missing even more than a decade after TrustZone was initially released [17]. One root cause for the lack of progress in TrustZone-based TEEs' adoption is that each installed Trusted App (TA) increases the potential for security-critical vulnerabilities, allowing attackers to exploit bugs and escalate privileges, exfiltrate private data, or gain complete control over the entire device. In practice, this means that bugs in TrustZone-enabled applications expose a large number of devices to real-world security threats, as continuously demonstrated by security researchers across device families and hardware vendors (e.g. in 2014 [15], [32], in 2015 [19], [47], in 2016 [20], [49], and in 2017 [52], [44]). Google's ProjectZero [45] recently summarized the main flaws of the current design of TrustZone as follows: it combines (i) weak isolation between TAs in the TEE, with (ii) Trusted Computing Base (TCB) expansion, and (iii) highly privileged access to the platform, making TrustZone a high-value target for attackers. Thus, vendors often aim to control and restrict access to the TEE. Thorough security assessments are needed to build a trust relationship between device vendor and app developer. Furthermore, deploying TAs to a TEE produces a large management overhead [23]. For smaller developers, the emerging costs pose a significant investment severely limiting the development of secure mobile services in practice. Device vendors try to circumvent these problems by offering some TEE functionalities, e.g. key storage, over interfaces to normal-world apps. However, this approach does not allow developers to protect own security-sensitive code and data. Hence, the provided TEE services are not sufficient to implement feature-rich secure mobile services.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23448
www.ndss-symposium.org

Existing Security Architectures. A number of ARM-based security architectures have been proposed previously [28], [10], [18]. However, they rely on virtual memory for isolation, using the same isolation mechanism proven insufficient for isolating TAs within ARM TrustZone's secure world [45]. Approaches that rely solely on *temporal isolation* – i.e., suspending the entire system to provide protection for TA execution – are not suitable for today's multi-core platforms [38], [51], since they effectively disable multitasking and parallel execution for the entire platform which imposes severe restrictions that directly affect user experience.

Goals and Contributions. Our main goal is to tackle the aforementioned problems and enable the full potential of TEEs for third-party application developers without requiring any hardware changes.

To this end, we present SANCTUARY, a novel security architecture for Trusted Execution Environments (TEEs) based on the latest ARM System-on-Chip (SoC) reference designs. SANCTUARY inherently de-privileges TrustZone-enabled apps by moving them from the secure-world TEE to an isolated normal-world compartment, thereby reducing the code base in the secure world. We call these security-sensitive apps, which are comparable to SGX's user-space enclaves, Sanctuary Apps (SAs). SANCTUARY achieves SA isolation by dynamically partitioning and re-allocating system resources: CPU cores and physical memory are temporarily reserved for the isolated compartments to execute SAs without suspending the rest of the system. In particular, we leverage TrustZone's Address-Space Controller (TZASC) to ensure a hardware-enforced, two-way isolation between SAs and all other system components. This enables an SGX-like usage of TrustZone without requiring any hardware modifications.

Building SANCTUARY comes with a number of interesting challenges: first, the Legacy OS normally assumes full control over all available CPUs. To support dynamic re-allocation of cores we have to claim, initialize, and boot individual cores dynamically at run time. Second, enforcing a strict separation between normal world, SAs, and secure world necessitates communication channels between them, e.g., to relay I/O or shared data. Third, SANCTUARY must provide security services, such as *remote attestation* and *sealing* of SAs (similar to SGX), and provide secure ways for SAs to access them. Finally, to offer tangible improvements in real-world scenarios, SANCTUARY must provide adequate performance, e.g., in authentication for mobile banking applications, without affecting user experience. Our design of SANCTUARY tackles all of these challenges to support SGX-like usage of TrustZone-enabled applications.

To summarize, our main contributions are as follows:

- We present the design of SANCTUARY, a novel security architecture building on existing TrustZone's hardware and software components while enabling enclave-like usage in the form of de-privileged normal-world execution environments that are completely isolated from the rest of the system.
- Our proof-of-concept implementation of SANCTUARY uses the HiKey 960 development board, and Linaro's open-source software OP-TEE on top of TrustZone.
- We analyze and discuss the security properties of SANCTUARY in a strong adversary setting that includes malicious SAs.
- We extensively evaluate SANCTUARY with respect to its setup and communication overhead. Additionally, we demonstrate real-world benefits of SANCTUARY in a detailed one-time password and key-generation use case for two-factor authentication, which is highly relevant for many security-sensitive applications such as mobile payment. Our results show that SANCTUARY supports low latency and does not affect user experience, hence, offering practical performance characteristics.

II. BACKGROUND

The core principle of TEEs is isolation of code and data to protect their integrity and confidentiality.

TEEs have been developed by both, academic community and industry. First, we present ARM TrustZone [5] which is available on most ARM-based systems and which is the basis for our novel security architecture SANCTUARY. Second, we explain the TrustZone Address Space Controller (TZASC) that enforces memory access control in TrustZone and plays a key role for our hardware-based isolation in SANCTUARY.

We discuss TEE research proposals as well as other related approaches in detail in Section VIII.

A. ARM TrustZone

TrustZone represents a set of security enhancements to processor designs and SoCs that are based on the ARM architecture. TrustZone enhances the processor, memory (including caches), and peripherals. A TrustZone-enabled processor can execute instructions in four different privilege levels (*Exception Levels* – EL0-EL3) and, additionally, two security modes at any given time (cf., *normal world* and *secure world* in Figure 1). To facilitate switching between normal and secure world, and to provide a clean interface, EL3 (also called *monitor mode*) runs the ARM Trusted Firmware (TF). On top of the Trusted Firmware (TF), the secure and normal world both manage their own address spaces using the remaining privilege levels for separation: EL2 is optionally used for a hypervisor, EL1 for the OS kernel, and EL0 (lowest execution privilege) is used for execution of application code.

The processor can switch from normal to secure world via an instruction called the secure monitor call (`smc`). When an `smc` instruction is invoked from normal world, the processor-core performs a context switch to the secure world (via the monitor mode) and freezes its normal-world execution. All other CPU cores of a multi-core system can independently remain in normal-world mode.

TrustZone can separate physical memory into two partitions, with one partition being exclusively accessible by the secure world. This isolation is enforced by the memory controller (TZASC), which is discussed in Section II-B. While the normal world cannot access memory assigned to the secure world, the secure world can access normal-world memory.

A device running ARM TrustZone boots up in the secure world. After the secure world finished its initial setup by

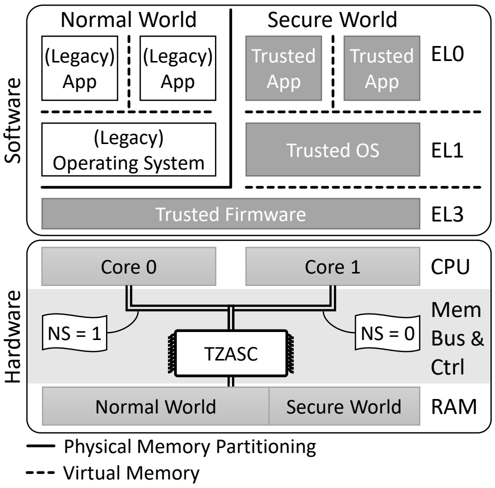


Figure 1: TrustZone software and hardware components. Software can be executed in *normal world* or in *secure world*. Isolation between these two worlds is enforced by the memory controller (TZASC) that checks for each memory access which world it originates from.

booting the Trusted OS (TOS), it switches to the normal world and boots the LOS. Most TrustZone-enabled devices are configured to use *secure boot*, i.e., the boot loader cryptographically checks the TOS prior to execution [5]. In fact, many vendors lock their devices against end-user modification via secure boot, to ensure integrity of the secure world. This allows them to make the secure world part of their TCB.

B. TrustZone Address Space Controller

With TrustZone, secure-world memory is isolated from the untrusted normal-world memory through physical memory partitioning. This is enforced in hardware by the TrustZone Address Space Controller (TZASC) which resides between the system bus and the memory chip (see Figure 1). It supports multiple memory regions and access-control settings based on several bus transaction characteristics. Originally, this only included two types of memory accesses: *non-secure* access ($\texttt{NS} = 1$), or *secure* access ($\texttt{NS} = 0$). A CPU core in *secure* mode can perform accesses of the type *secure* and *non-secure*, whereas CPU cores in *normal* mode can only perform *non-secure* accesses. The first TZASC reference implementation from ARM, the TZC-380, was published in 2010 [6]. Its successor, the TZC-400 [8], was introduced 2013 and can utilize additional characteristics of a bus transaction to separate the protected memory regions – this feature is called *identity-based filtering*. Thus, in current ARM reference designs, every device that can act as a bus master (e.g., CPU, GPU, DMA controller) is assigned a *bus-master ID* in hardware, which is appended to its memory bus transactions. This can be used to assign memory regions to specific bus masters for non-secure accesses. ARM advertises the identity-based filtering feature in context of their TrustZone Media Protection Architecture (TZMP) [4], which is used for media protection by exclusively assigning memory, e.g. the frame buffer, to the GPU.

III. Adversary Model and Requirements

A. Adversary Model

Our threat model adheres to that of TrustZone and makes the same underlying assumptions [5]. In particular, the attacker can corrupt all normal-world software, including all privilege levels up to an optional hypervisor (EL2), via remote or local software attacks. Additionally, an adversary can conduct passive physical attacks. However, the adversary cannot compromise the secure-world software and the monitor mode.

Invasive physical attacks that tamper with hardware, e.g., to inject faults at run time are out of scope. Similar to TrustZone, we do not consider Denial-of-Service (DoS) attacks,i.e., Sanctuary does not provide availability guarantees.

Our detailed standard assumptions are derived from the related work [22], [10], [9], [12], [16], [28]:

- Applications in normal world are considered untrusted.
- The Legacy OS (LOS) in the normal world is untrusted.
- Isolation between different privilege levels is enforced by hardware through virtual memory.
- All existing architectural defenses, such as Execute Never (XN), Unprivileged Execute Never (UXN), Privileged Execute Never (PXN), and Privileged Access Never (PAN) are deployed and active.
- Secure and normal world are isolated by the TrustZone hardware extensions [5].
- Software in the secure world, including the boot loader and EL3 firmware (monitor mode), is trusted.

In this setting, Sanctuary can be used to minimize the amount of software required in the secure world as it allows to outsource *all* Trusted Apps (TAs) to Sanctuary Apps (SAs) which execute in isolated compartments in the normal world.

B. Requirements Analysis

To enable practical and secure Sanctuary Apps (SAs) on ARM TrustZone-based platforms, a number of requirements must be fulfilled. We show that Sanctuary fulfills these security requirements in Section VI, and demonstrate that Sanctuary meets the functional requirements in Section VII.

1) **Code and data integrity.** The integrity of the code and data of an SA must be preserved. This can be achieved by (i) isolation during SA execution and (ii) attestation of the SA code when loaded into the isolated compartment.
2) **Data confidentiality.** Confidentiality of data processed in an SA must be preserved. This can be achieved by (i) a secure channel for provisioning the data, (ii) spatial isolation during execution, and (iii) temporal isolation to prevent that sensitive information becomes accessible after SA execution has finished.
3) **Secure channel to secure world.** An SA needs a secure channel to utilize security services provided by

the secure world. This can be realized by an *exclusive* shared memory, i.e., accessible only by the SA and the secure world but not by untrusted normal-world software.

4) **Protection from malicious SAs.** To enable unrestricted usage models for SAs, malicious SAs must be tolerated. Protecting the platform from malicious SAs can be achieved by limiting the access privileges of SAs to a minimum (i.e., EL0) and preventing them from accessing normal-world memory.
5) **Hardware-enforced resource partitioning.** To ensure strict isolation spatial *and* temporal isolation are needed.
6) **Minimal software changes.** Leveraging existing interfaces of the secure-world OS and the normal-world OS prevents extensive modification of the software stack.
7) **Positive user experience.** Assigning a single CPU core for limited time to SA execution leads to low impact on the overall system performance for most usage scenarios on todays commonly available multi-core architectures. Latency can be kept low by minimizing the SA run-time environment.

IV. Sanctuary Design

The goal of the Sanctuary architecture is to enable secure and widespread use of Trusted Execution Environments (TEEs) (e.g., through third-party developers) on ARM based devices. Sanctuary allows the creation of multiple parallel isolated compartments on ARM devices in the normal world which are strictly isolated from the LOS and Legacy Apps (LAs). The isolated compartments, which we call Sanctuary Instances, run security-sensitive apps called Sanctuary Apps (SAs). Every Sanctuary Instance executes only one SA at a time. Since all Sanctuary instances are independent and separated from each other, also the SAs become strongly isolated. Additionally, all Sanctuary instances are isolated from the existing TrustZone secure world.

Spatial isolation of a Sanctuary Instance is achieved by (i) partitioning the physical memory using the TZC-400 memory controller, (ii) dedicating a CPU core to the Sanctuary Instance, and (iii) excluding the Sanctuary Instance's memory from shared caches. Temporal isolation is ensured by launching the Sanctuary CPU core from a trustworthy state (ARM Trusted Firmware (TF)) and erasing all sensitive information from memory and caches before it exits.

We designed Sanctuary in such a way that the required changes to the existing software ecosystem are *minimal*: in fact, Sanctuary can extend existing TEE architectures without affecting the functionality of already deployed software in both the normal world and the secure world.

Figure 2 shows an abstract view of Sanctuary's design. In the following, we describe Sanctuary's isolation mechanism, its initialization, and its security services.

A. Sanctuary *Isolation*

In addition to the existing security boundary between TrustZone's secure world and normal world, Sanctuary enables isolation *within* the normal world. A dedicated memory region is made exclusively accessible by one CPU core by leveraging ARM's new memory access controller TZC-400. Details on how the controller needs to be configured to achieve this physical memory partitioning are given in Section V-E. As a result, all software executing on that CPU core is protected from untrusted software executing on the remaining CPU cores of the system. In Figure 2, CPU core 2 running a Sanctuary Instance is configured to have exclusive access to the Sanctuary *RAM* partition, as depicted by the arrows. The untrusted normal-world software – executing on CPU cores 0 and 1 – can only access the normal world memory. Furthermore, the CPU core assigned to the Sanctuary Instance is *not* allowed to access normal-world memory, achieving a two-way isolation which allows Sanctuary to tolerate potentially malicious SAs. However, Sanctuary does support shared memory between normal world and SA for efficient communication as well as shared memory between secure world and SA to establish a secure channel. This enables scenarios like secure UI over TAs. Sanctuary's handling of shared memory is explained in detail in Section V-E.

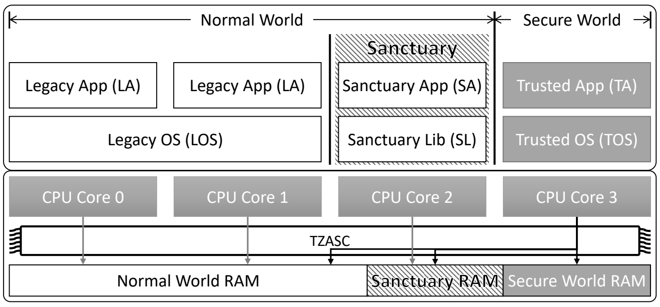


Figure 2: Sanctuary design overview. Within the normal world, one core is reserved for Sanctuary. The TCB, marked in gray, includes the hardware and the secure-world software that is involved in the initialization of an SA.

The secure-world software is trusted and therefore allowed to access all memory, including normal-world memory, Sanctuary memory, and secure-world memory (black arrows in Figure 2).

Multi-SA isolation: Sanctuary instances are either executed consecutively on the same CPU core, or execute on separate, mutually isolated cores with dedicated memory partitions. After SA execution finished, the system returns to its original state (see Section IV-B) and the next Sanctuary instance can be launched. This ensures strong isolation between SAs: all SAs are executed completely independently of each other.

Privilege isolation: SAs are limited to execute in user-mode. The privileged mode of a CPU core used by Sanctuary is occupied by the Sanctuary Library (SL). Important to note is that the SL is *not* part of the TCB, but instead is only needed to provide two main functionalities: (i) initializing an execution environment for the SA, and (ii) providing service interfaces to the SA, e.g., for accessing Sanctuary's security services.

B. Sanctuary *Initialization*

Sanctuary's isolation does protect the integrity and confidentiality of an SA while it is executing on the dedicated CPU core. However, since the SA code is loaded by the untrusted LOS, its integrity must be verified. The initialization process of SANCTUARY provides the necessary verification mechanism.

For better resource utilization, SANCTUARY does not dedicate one CPU core for executing SAs permanently. If a new SANCTUARY instance is created, one CPU core is shut down and removed from the resources available to the LOS executing in the normal world. All remaining CPU cores stay under control of the LOS. Hence, the LOS can continue execution of normal-world tasks preserving the system's availability, i.e., the user does not notice negative effects from the creation of a SANCTUARY Instance and the execution of an SA.

Next, the code to be executed on the SANCTUARY core, i.e., SL and SA, is loaded into a separate memory section. After the memory isolation has been activated, the loaded code is validated using digital signatures. The signature for the SL is provided by the device vendor, whereas the signature for the SA is provided by the SA developer. The detailed verification process is described in Section V. After a successful verification, the dedicated CPU core is restarted. The SANCTUARY core starts from a defined initial state, boots the SL and executes the SA.

After an SA has finished, the dedicated core removes all information from the memory, invalidates all cached data, and shuts down. The isolation for the wiped memory is deactivated, making the memory available to the LOS again. The CPU core is restarted and reassigned to the LOS.

C. SANCTUARY *Security Services*

The initial content of an SA is loaded from unprotected memory, hence, it can be manipulated and cannot contain confidential data. Therefore, SANCTUARY needs to provide a mechanism to provision confidential data to an SA over a secure channel *after* it has been created. However, to ensure that secret data is not sent to a malicious (or maliciously modified) SA, the integrity and authenticity of an SA needs to be verified before provisioning secret data. To enable secure provisioning of secret data to an SA and secure storage of secret data, SANCTUARY provides a set of security services implemented as TAs supplied by the device vendor (called vendor TAs throughout the remaining paper). These TAs run within the secure-world Trusted OS (TOS) (see Figure 2).

Remote attestation allows an SA to establish a secure channel to an external entity. Through the platform identity feature of TrustZone, the integrity measurement of SANCTUARY can be authentically reported to a third party. Linking the authentic integrity report with the establishment of a secure channel to the SA creates a secure and authenticated channel through which confidential data can be provisioned.

Sealing allows SAs to store sensitive data such that only instances of the originating SA can accesses the data. SANCTUARY provides each SA with a unique encryption key that is derived from the hash value computed over the SA binary. The key can be used to encrypt data, e.g., before writing it to persistent storage.

Further security services, like monotonic counters, secure timers, secure randomness, etc. can be provided by TrustZone's secure world, as well. Similar security services are commonly available in commercial TEE implementations, for instance Intel SGX [31], [39], [25], [2] and can be implemented similarly in SANCTUARY. In addition, secure user interfaces for SAs can easily be provided by TAs, as secure I/O is already provided by TrustZone.

D. SANCTUARY *Software Model*

With SANCTUARY, every application developer is able to utilize TEE functionalities, i.e., every developer can deploy an SA. Each SA belongs to an untrusted LA. This allows straightforward deployment through existing app markets: SAs come as part of LAs using the standard installation routine.

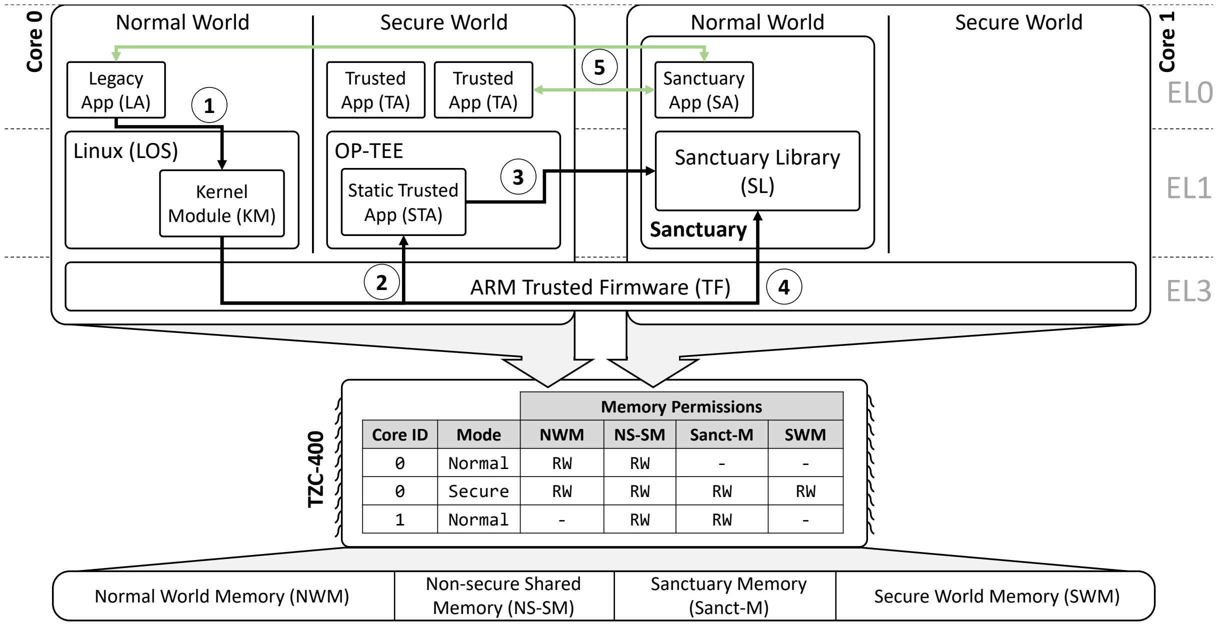
Additionally, by coupling each SA with an LA, the functionalities of the SL can be minimized. In particular, the LA acts as a proxy and allows the SA to make use of all functionalities provided by the LOS, like file system access. The LA and SA can efficiently exchange information and interact with each other via shared memory. When an SA wants to provide sensitive data to the LA, e.g., for persistent storage, the SA can use the sealing service (see Section IV-C) to encrypt the data before sending it to the LA.

How to partition an application into security-critical and uncritical parts is an orthogonal problem.

V. IMPLEMENTATION

System Setup. We implemented SANCTUARY on a HiKey 960 development board, as it provides a recent ARMv8 SoC design that is commonly used on modern mobile devices. Moreover, the HiKey 960 is one of the few development boards which gives developers the possibility to deploy own software in the secure world. The HiKey 960 is based on an octa-core ARM big.LITTLE processor architecture with four ARM Cortex-A73 and four Cortex-A53 cores.

SANCTUARY Software Components. An overview of our SANCTUARY implementation is shown in Figure 3. For the secure-world Trusted OS (TOS) we use OP-TEE [1] which currently is the most developed open-source TOS. The SANCTUARY design is not limited to a particular TOS and can also be implemented using a TOS which provides a less rich feature set. OP-TEE comes bundled with a recent Linux distribution which we use as the normal-world Legacy OS (LOS). We implement a custom kernel module (KM) as part of the LOS which manages the SANCTUARY Instances from the normal world. In OP-TEE, we implement two vendor Trusted Apps (TAs), the *Proxy TA* and the *Sealing TA*. They provide the basic security services for SANCTUARY, namely remote attestation and sealing. A SANCTUARY Instance consists of the Sanctuary Library (SL) and a SA. In our prototype implementation, we use the Zircon micro kernel [24] as the basis for our SL. Besides adding two vendor TAs, we only make small one-time changes to the Trusted Computing Base (TCB), i.e. OP-TEE and the ARM TF. The custom Static Trusted App (STA) which we add to OP-TEE manages the SANCTUARY Instances from the secure world. The Lines of Code (LOC) added to the TCB add up to 1313. The two vendor TAs make up more than half of the added lines. In total however, the TCB gets reduced because all TAs from third-party developers are removed from the secure world.



		Memory Permissions			
Core ID	Mode	NWM	NS-SM	Sanct-M	SWM
0	Normal	RW	RW	-	-
0	Secure	RW	RW	RW	RW
1	Normal	-	RW	RW	-

Figure 3: Implementation overview of SANCTUARY.

Since no source code of third-party TAs used on current devices is publicly available, we refer to Huang et al. [29] for average TA sizes. They implemented a mobile payment and chat TA consisting of 900 LOC and 200 LOC, respectively, which can be seen as a lower limit for implementing a useful TA. This shows that removing all third-party TAs from the secure world outweighs additions made to OP-TEE and the Trusted Firmware (TF) by an order of magnitude in terms of LOC. The number of added or modified LOC for all components are shown in Table I.

SANCTUARY Hardware Components. In SANCTUARY, we utilize the fact that unique master IDs can be assigned to every CPU core and therefore also to every memory transaction performed by a core. These transaction IDs can then be used to filter memory accesses on a hardware level. As such, memory regions can be made core-exclusive. The filtering and permission enforcement is performed by the TZC-400 memory controller. The TZC-400 allows or denies access to memory regions depending on two properties: (i) the type of the access transaction performed by the core running the code (*secure* or *non-secure*), and (ii) the bus master ID of the core which executes the SANCTUARY Instance. Enforced access permissions are shown in Figure 3 in the *Memory Permissions* table, and are described in detail in Section V-E.

SANCTUARY Usage. The high-level SANCTUARY life cycle works as follows: when a LA wants to execute sensitive code in form of an SA inside a SANCTUARY Instance, the LA requests execution of its bundled SA from the KM ①. The KM initiates the setup of the SANCTUARY Instance by loading the SANCTUARY binaries (SL and SA). Next, the KM removes one CPU core from the LOS and hands over control to the STA ② to perform all security-related steps, such as the verification of the SA ③. After successfully setting up of the SANCTUARY Instance, the KM triggers the SANCTUARY boot ④. When the boot process is finished the SA can execute the sensitive code, and communicate with its LA as well as with the TAs in the secure world ⑤.

Component	World	Added LOC	Modified LOC
Kernel Module	normal	713	-
Zircon Micro Kernel	normal	166	45
OP-TEE	secure	56	2
Static Trusted App	secure	472	-
ARM Trusted Firmware	secure	92	-
Proxy TA	secure	287	-
Sealing TA	secure	406	-

Table I: Modifications for Sanctuary Components.

In the following, we explain each component of SANCTUARY and its life cycle in detail.

A. Legacy OS

With SANCTUARY, the resource management remains in the LOS. We implement the required functionalities in a custom loadable kernel module (KM). The KM manages all resources needed for a SANCTUARY Instance. It is able to remove a core from the LOS and also to hand the core back to the LOS after a SANCTUARY Instance finished execution. Since we use Linux as the LOS in our prototype, we utilize the Linux CPU hotplug mechanism [11] for that purpose. Furthermore, the KM dynamically allocates memory for SANCTUARY Instances and their associated communication channels from SA to LA and from SA to TAs. Before a SANCTUARY Instance can be started, the KM has to load the SANCTUARY binaries (SL and SA) into RAM which will be exclusively assigned to the SANCTUARY core afterwards. The OP-TEE driver facilitates the communication between LOS and OP-TEE.

B. Security Services

We keep the traditional structure of the secure world: Trusted Apps run on SEL0 (secure-world user space), while OP-TEE runs on SEL1 (secure-world kernel space). The TAs offer relevant security services. In our proof-of-concept implementation, we implemented a *Proxy TA* and a *Sealing TA*. The *Proxy TA* is used to establish a secure communication channel from an SA to remote servers. All data sent through the *Proxy TA* is authenticated with the platform key and bound to the identity of sender SA, i.e., the Proxy TA provides **remote attestation**. The *Sealing TA* provides **sealing** functionality which allows to bind data to a specific SA and to store it permanently on the device. For each SA an individual key is used.

A Static Trusted App (STA) represents a kernel module in OP-TEE. In our prototype, the STA verifies the SL using a preconfigured signature, sets up the SANCTUARY Instances, and tears them down. Moreover, the STA provides functionalities to TAs which can be used to, e.g., find out which SA is currently running in a SANCTUARY Instance, or to compute a hash over an SA binary.

All aforementioned security services rely on the Trusted Firmware (TF) as a trust anchor which is responsible for context switches between normal and secure world and low-level platform services. In our prototype, the TF was extended to verify several security-relevant steps during the SANCTUARY life cycle which is explained in detail in Section V-E.

C. Sanctuary

We implemented isolated code execution in SANCTUARY by running SANCTUARY Instances in the normal world on dedicated CPU cores. This isolates SANCTUARY Instances from untrusted LOS and TAs running on the remaining cores. A SANCTUARY Instance consists of two parts: the SL and an SA. The SL provides basic process and memory management functionalities for running an SA. In our implementation, we chose the Zircon micro kernel [24] as SL due to its small size (approx. 1MB) and versatility. After Zircon boots, it prepares the environment for the SA by configuring the CPU core, setting up the memory mappings and a basic execution environment. Then, the SA is started as a normal-world user process by the Zircon micro kernel. During execution, an SA can communicate with its corresponding LA and also with TAs in the secure world to utilize their provided security services (e.g., sealing or remote attestation). To achieve this, we extend the Zircon micro kernel with new system calls.

As required by SANCTUARY, the STA prevents simultaneous execution of SAs in one SANCTUARY Instance because this could lead to sensitive information leakage between SAs.

D. Memory Isolation Unit

In addition to isolating SANCTUARY execution through dedicated CPU cores, we protect SANCTUARY memory against normal-world accesses from other cores by leveraging the ARM TrustZone Address Space Controller (TZASC). As described in Section II, its recent implementation, the ARM TZC-400, allows setting memory-access permissions based on bus master IDs. Traditionally on ARMv8 architectures, all cores already have uniquely-assigned multi-processor-IDs (MPID register [3]). For all transactions sent to the system bus, multi-processor IDs are then translated to bus master IDs by a dedicated labeling component. Currently, as on the HiKey960 development board, transactions from all cores are labeled with the same bus master ID. For Sanctuary, only the mapping policy needs to be changed such that the bus transactions of cores are labeled with unique bus master IDs. No hardware modifications have to be made to the processor-core. We implemented the modified labeling ID-mapping policy using the ARM Fast Models virtualization tools. From software, we can now configure the TZC-400 such that memory regions can be exclusively assigned to single cores by filtering the bus transactions for the buster master ID labels. Details on how the TZC-400 needs to be configured are given in Section V-E.The performance overhead for configuring the TZC-400 is negligible compared to the rest of the Sanctuary startup, it only consists of a few register writes. It is important to mention that the assignment of bus master ID labels to transactions is already performed on all transactions by default. We only enforce the labeling of unique IDs. This means on the hardware level, SANCTUARY produces zero performance overhead. Therefore, evaluation on a Hikey 960 board gives realistic performance measurements.

If not enough unused bus master IDs are available to distinguish all core transactions, only a subset of the cores can run Sanctuary instances. This does not limit the general applicability of Sanctuary as long as at least two free bus master IDs are present.

On systems with the TZC-400, no additional hardware components are needed to implement SANCTUARY. Some device vendors already license the TZASC IP from ARM since it provides an industry-ready solution (e.g. Samsung on the Exynos chips [13]). Unfortunately, public information regarding the deployment of the TZC-400 on current platforms is limited.

E. Execution Life Cycle

In our prototype, a typical SANCTUARY life cycle consists of four phases: *(a) Sanctuary Setup*, *(b) Sanctuary Boot*, *(c) SA Execution*, and *(d) Sanctuary Teardown*, which we will explain in the following. In our prototype, we assume that a signature of the SL binary is already stored in the secure world. However, integrity and authenticity of the SL can generally also be established through certificates. Remote attestation of the SA can be achieved by leveraging the Proxy TA. However, alternative schemes, like Intel EPID, could be implemented as well. The implementation details of such a scheme are out of scope for this paper, thus, we refer the reader to Intel's documentation for a possible outline [36].

Sanctuary Setup. The SANCTUARY setup phase is performed by the KM in the normal world and the STA in the secure world. The KM manages system resources, whereas the STA performs all security relevant steps. The setup of a SANCTUARY Instance is triggered by the LA that requests execution of its sensitive code in the corresponding SA. Subsequently, the SL and SA binaries are loaded from the file system and handed over to the KM using *procfs*. The SL binary can also be loaded only once during system boot and remain in memory until the system is shut down. We implemented the binary loading

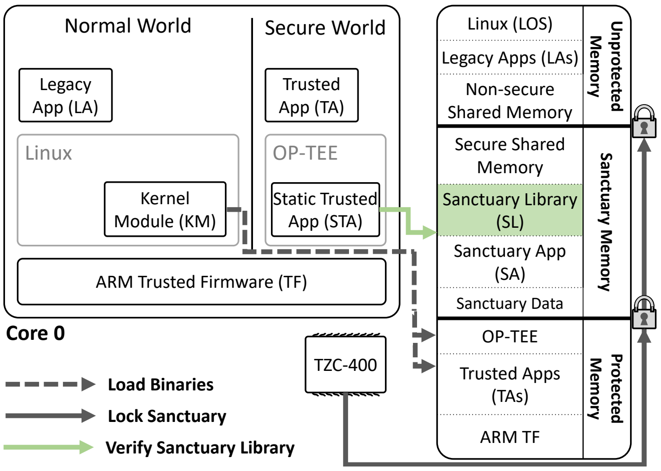


Figure 4: Memory layout after setting up a Sanctuary. As before, **Core 0** is responsible for initializing and managing the Sanctuary App, which runs on **Core 1** (omitted for clarity).

in the normal world since OP-TEE cannot directly access the file system. The KM reserves additional memory for the Sanctuary Instance which is used for memory allocations during SA run time. We call this memory area *Sanctuary Data*. Additional memory is reserved for the Sanctuary Instance's communication channels. The communication between the LA and its SA is performed over non-secure (i.e. accessible by untrusted software) shared memory. In contrast, secure shared memory is used for communication between SA and TAs. The final memory layout after the setup is depicted in Figure 4.

After loading the binaries, the KM selects a CPU core to run the Sanctuary Instance. The KM always selects the CPU core with the least load. Next, the Linux hotplug mechanism is used to shut down the selected core. If successful, the KM calls the STA and provides the ID of the selected core as an argument. This call traps into monitor mode where the TF checks that the selected core is indeed shut down before performing a world switch and handing over control to the STA. The STA then locks the Sanctuary memory by configuring the TZC-400.

We assume that unique IDs `0-7` are assigned in hardware to 8 CPU cores, and that the selected Sanctuary core has the ID `7`. Moreover, for the sake of simplicity, we assume that no other bus master than the CPU needs access to memory. Then, one of the up to 9 memory regions the TZC-400 can separate is configured to exactly cover the contiguous memory area in which the SL and SA binaries, the Sanctuary Data and the secure shared memory resides. We assume that region `1` is used for that purpose. The lowest address covered by region `1` is set using the *REGION_BASE_LOW_1* and *REGION_BASE_HIGH_1* registers. The highest address covered is set using the *REGION_TOP_LOW_1* and *REGION_TOP_HIGH_1* registers. Subsequently, the configured memory region `1` is solely assigned to the Sanctuary core using the *REGION_ID_ACCESS_1* register. The bit assignments of the region ID access register is shown in Figure 5. The upper 16 bits of the register define the non-secure write access permissions (nsaid_wr_en), the lower 16 bits the non-secure read access permissions (nsaid_rd_en). Every bit is associated with one bus master ID. This means, if e.g. bit 0 and bit 16 of *REGION_ID_ACCESS_1* are set to `1` and all other bits to `0`, only the bus master with the associated ID `0` is allowed to perform write or read access on region `1`. In our scenario, the Sanctuary memory is assigned to the Sanctuary core by setting *REGION_ID_ACCESS_1* to the value `0x800080`. Then, non-secure access is only allowed for the core with ID `7`, which is the Sanctuary core in our example. For the memory regions that cover all of the normal-world memory except the non-secure shared memory, the bits are set to `0x7F007F` in the corresponding *REGION_ID_ACCESS* registers. Thus, permission to access the normal-world memory is granted to all cores except the Sanctuary core. This is crucial for implementing two-way isolation. The region covering the non-secure shared memory is configured with the value `0xFF00FF` since the cores running the normal world and also the Sanctuary core need access to it. As a last step, the regions covering the secure-world memory are configured with the bit value `0x0` such that no core can perform a non-secure access on the memory.

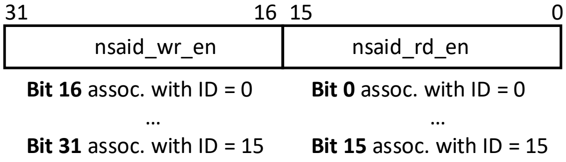

31
16
15
0
nsaid_wr_en
nsaid_rd_en
Bit 16 assoc. with ID = 0
...
Bit 31 assoc. with ID = 15
Bit 0 assoc. with ID = 0
...
Bit 15 assoc. with ID = 15

Figure 5: Region ID Access Register.

The resulting memory permissions are listed in Figure 3 for the different memory regions, core IDs, and execution modes. In the following verification step, the STA verifies the SL binary using the stored digital signature. For this purpose, the STA uses the RSASSA-PKCS1-v1_5 scheme together with SHA-256 which are provided by OP-TEE. After successful verification, the ARM TF is informed that the Sanctuary is locked, verified, and ready to be booted.

Sanctuary Boot. After successful Sanctuary setup, the KM calls the TF to boot the Sanctuary core. Before starting the core, the TF checks that the Sanctuary Instance was correctly locked and verified. After receiving the boot signal, the Sanctuary core first executes the TF in EL3. During initialization of the TF, exception handlers needed for calling the TF from the SL are set up. The TF needs to be callable from the SL to shut the Sanctuary core down in the teardown phase. After TF initialization, the core switches to EL1 and jumps to the entry point of the SL. We slightly modified the Zircon boot sequence to prevent information leakage from the Sanctuary Instance We slightly modified the Zircon boot sequence to prevent information leakage from the Sanctuary Instance by excluding all Sanctuary memory from being cached in the shared L2 cache. Moreover, the external interrupts are configured using the core's CPU interface of the General Interrupt Controller (GIC) which cannot be accessed by other cores. This blocks external interrupts triggered by other cores, while allowing to receive interrupts requested by the Sanctuary core, e.g. timer interrupts.

SA execution. While executing sensitive code, the SA may establish communication channels. The SA is able to communicate with its corresponding LA over the non-secure shared memory and with vendor TAs over the secure shared memory through OP-TEE. All data sent over the non-secure shared memory is accessible to the normal world, and hence, it is not part of the SANCTUARY memory partition. Communication is facilitated by the KM of the LOS in the normal world and by custom Zircon system calls on the SANCTUARY side. When the SA requires security services from vendor TAs, it communicates with the TA over the secure shared memory channel. On the secure-world side, this communication is facilitated by the STA. Since all data shared between the SA and a TA is sensitive, this data is solely exchanged over secure shared memory which is part of the protected SANCTUARY memory region assigned to the SANCTUARY core.

SANCTUARY allows two different implementation variants for SA to TA communication: (i) the OP-TEE driver is included in Zircon and the world switch to the secure world is performed by the SANCTUARY core itself, and (ii) the connection to the secure world is triggered by the SA's corresponding LA. This means an SA first has to communicate with the normal world before it can communicate with a TA. However, the data exchanged between TA and SA remains unaccessible to the LA. We implement the second variant in our prototype since it requires less modifications to the Zircon kernel.

Sanctuary Teardown. The three-step teardown of the SANCTUARY is triggered by the LA. The first step is to shut down the SANCTUARY core when the LA signals to the SA that its services are not needed anymore. Subsequently, the SA saves its state (if needed) using e.g. the sealing services. Next, internal clean up actions bring the Zircon kernel back in its original state and invalidate the L1 cache to prevent data leakage. Then, the SL signals the STA that it successfully performed the clean up action. Subsequently, the TF is used to shut down the core. The second step is to unlock the SANCTUARY memory. Analogously to the locking in the setup phase this is performed by the STA. Again, the modified TF checks that the SANCTUARY core is indeed shut down before performing the world switch and handing control over to the STA. The STA checks if the SA was able to perform its clean up action. Then, the secure shared memory and Sanctuary Data memory are zeroed to prevent leakage of SA data. Finally, the configuration of the TZC-400 is reverted such that the SANCTUARY memory region and the SANCTUARY core are freed. In the third step of the teardown process, the KM uses the Linux hotplug mechanism to reclaim the available core.

VI. SECURITY ANALYSIS

The goal of SANCTUARY is to protect against a strong attacker, as described by our adversary model (see Section III-A). We also derived the requirements for our design of SANCTUARY in Section III-B. For a systematic analysis of SANCTUARY, we will now look at all possible attack vectors that are available to an adversary in our threat model. In particular, we can see from Figure 2 that attacks can originate from three different locations on the platform: (i) the normal-world user space, (ii) the normal-world OS, and (iii) a malicious SA. In all three cases, the goal of the attacker is to compromise the integrity or data confidentiality of a victim SA or gain control over the LOS. This can happen at any point in time during the life-cycle of an SA (i.e., setup, boot, execution, or tear-down), and hence, we will discuss each case in the following. In particular, malicious code in the normal world can aim at either manipulating the SL and SA binaries before they are loaded (Section VI-A), overcome the isolation of SANCTUARY (Section VI-B), manipulate persistently stored data of an SA (Section VI-C), or extract information from an SA via the cache (Section VI-D). We discuss the case of malicious SAs in Section VI-E. As we will show, an adversary cannot compromise the security of SANCTUARY in any of those cases, and does not gain any advantage from executing code inside an SA over regular normal-world execution.

A. Binary Integrity

SL's and SAs's binaries are saved unencrypted in normal-world memory. Nevertheless, SANCTUARY ensures integrity of these binaries by using local attestation and by providing functionalities for remote attestation, respectively. SANCTUARY stores a signature of the SL in the secure-world memory. Before a SANCTUARY Instance is started, the STA performs a local attestation by measuring the SL binary and by verifying it against the stored signature. If verification fails, SANCTUARY's setup is aborted and the modified code therefore never executed. Developers can verify an SA's integrity using remote attestation. Whenever an SA connects to a server, TEE functionalities are used to establish a secure connection to the server. Moreover, the STA creates a signature of the SA which is also send to the server. Thus, the server can check if the SA is in a valid state before provisioning sensitive data to it. These properties, together with the properties in Section VI-B, fulfill security requirement 1: *Code and data integrity*.

B. Code and Data Isolation

The SANCTUARY design provides strong hardware-enforced isolation of code and data. The SANCTUARY memory isolation is enforced by TrustZone before the integrity of the SL is verified. Once the SANCTUARY memory is locked, no core except the selected SANCTUARY core can perform non-secure reads or writes on the SANCTUARY memory region. The selected SANCTUARY core always boots in the TF and then jumps to an address in the SL which is set as a constant in the TF. During the boot process of the SANCTUARY Instance, the SL ensures that all interrupts from the system-wide interrupt controller, triggered from other cores than the SANCTUARY core, are disabled. Only a core itself can configure its interface to the GIC. Therefore, the execution of a SANCTUARY Instance cannot be interrupted by another core. Moreover, only the SANCTUARY core can shut itself down. During runtime, the SANCTUARY design makes sure that sensitive data is only passed to and received from a locked SANCTUARY Instance. When performing a world switch to the secure world, the TF verifies that the call was issued from the SANCTUARY core. Access from all other cores to the trusted functionalities in the TEE will be blocked. If the call was issued from the SANCTUARY core, the vendor TAs in the TEE use the STA to check if the SANCTUARY Instance is in correct state before reading or writing any data to the memory shared between secure world and SA. SANCTUARY also prevents the injection of data into the free SANCTUARY memory space before a SANCTUARY Instance is locked and the extraction of data after a SANCTUARY Instance is unlocked. The secure-world

STA overwrites SANCTUARY memory not reserved for either SL or SA with a fixed value after a SANCTUARY Instance is locked and before it is unlocked, including the secure shared memory. Besides, the SL is reset to its original state during shutdown, hence, it will not contain last executed SA's data. These properties fulfill security requirement 1: *Code and data integrity* in combination with the properties in Section VI-A. Additionally, in combination with the properties in Section VI-C, security requirement 2: *Data confidentiality* is fulfilled. Moreover, SANCTUARY's temporal and spatial hardware-enforced isolation fulfills security requirement 5: *Hardware-enforced resource partitioning* (in combination with the properties in Section VI-D). Finally, the exclusive shared memory between a SANCTUARY Instance and the secure world fulfills security requirement 3: *Secure channel to secure world.*

C. Secure Storage

SANCTUARY allows the secure and persistent storage of SA data using the STA and security services from the secure world. SANCTUARY ensures that the data is sealed to a SA entity using keys that are derived from a hash value computed over the SA binary. As a result, only an unmodified SA can successfully decrypt its own data. For the persistent storage of the sealed data, a SANCTUARY Instance uses the functionalities provided by the TEE. Depending on the TEE implementation, this might also allow the SA to bind its data to the device or to save it in roll-back protected memory. These properties fulfill security requirement 2: *Data confidentiality* (in combination with the properties from Section VI-B).

D. Cache Attack Resilience

As shown by recent Spectre [33] attacks, cache-based attacks can be very powerful. An attacker could, for instance, try to mount a software side-channel attack to extract data from cache lines used by a SANCTUARY Instance. Thus, these attacks are considered in SANCTUARY's design and implementation. As usual on ARMv8 platforms, we assume presence of first-level cache (L1) and second level cache (L2). On these platforms, first-level caches (L1) are core-exclusive, while the L2 cache is shared. This configuration allows two attack scenarios: direct attacks, and side-channel attacks.

Direct Attacks. A privileged attacker in the normal world could map the SANCTUARY memory region into an attacker-controlled memory space. This could potentially give an attacker *direct* access to the cached data of a SANCTUARY Instance, even without the permission to read the main memory for this physical address. For the L1 cache, we prevent this by running a SANCTUARY Instance on its own core and by invalidating the L1 cache before a SANCTUARY Instance is shutdown and unlocked. For the L2 cache, there are two ways to prevent direct attacks. One way is to configure the SANCTUARY memory region as `outer non-cacheable`, whereas the outer domain is represented by all caches outside of a particular CPU core. As a result, the SANCTUARY memory is never cached in the shared L2 cache. In Section VII, we show that this still gives practical performance. Alternatively, changes to the caches could be made on the hardware level to extend the enforcement of identity-based filtering to the L2. This prevents an attacker from directly accessing cache lines uses by a SANCTUARY Instance. In both cases, an attacker could also not inject own malicious data into the L2 cache. On ARMv8, data caches are normally either *Physically Indexed, Physically Tagged* (PIPT) or *Virtually Indexed, Physically Tagged* (VIPT) [3]. This means cache lines are tagged using physical addresses in both configurations. Since the attacker cannot write to or read from the physical addresses of the SANCTUARY memory, the attacker can also not fill the cache for those addresses.

Side-Channel Attacks. An unprivileged attacker could mount side-channel attacks like Prime+Probe [43] or Flush+Reload [53] to leak data from L1 or L2 caches. For the L1 cache, this is prevented by running a SANCTUARY Instance on its own core, i.e. the attacker cannot measure accesses to the SANCTUARY core's L1 cache while it is running. To prevent measurements after shutdown, a SANCTUARY Instance invalidates its L1 cache before it is shut down and unlocked. For the L2 cache, implementing the identity-based filtering does not solve the cache-side channel issue. Thus, cache partitioning (or a similar approach) is needed to prevent leakage. We prevent side-channel attacks on the L2 cache by excluding SANCTUARY memory from L2, which yields practical performance (cf. Section VII).

E. Malicious Sanctuary App

One strength of SANCTUARY is that third-party developers can easily create and deploy own SAs. This, however, also allows attackers to create malicious SAs. If an user is tricked into installing such an SA, it will be executed as a valid SA in a SANCTUARY Instance. The attacker could then try to attack the normal world or secure world from such a malicious SA, hence, SANCTUARY must protect against malicious SAs. With a malicious SA, an attacker might attack the LOS and LAs running in the normal world. Yet, an SA only has user privileges (EL0), EL1 is controlled by the device vendor providing the SL. If the attacker is able to successfully perform a privilege-escalation attack and compromise the SL, the secure-world memory is still not accessible for the attacker since the SA runs in normal world. In particular, since only the SANCTUARY memory is assigned to the SANCTUARY core, remaining normal-world memory could still not be accessed. Only the non-secure shared memory to the LA (developed by the attacker anyway) would be affected. An attacker could try to use a malicious SA to leak data from either other SAs or from TAs. However, since the SANCTUARY design dedicates CPU cores to SAs one at a time, unintended information flow between SAs is prevented. These properties fulfill security requirement 4: *Protection from malicious SAs.*

VII. EVALUATION

We evaluate SANCTUARY by implementing a real-world use-case in our prototype and by thoroughly measuring the performance of all SANCTUARY components. As mentioned in Section V, the overall implementation minimizes TCB changes by adding less than 1400 LOC. Thus, SANCTUARY fulfills functional requirement 6: *Minimal software changes* from Section III-B. The evaluation was performed on the HiKey 960 development board. The HiKey 960 provides an ARMv8 SoC design with an ARM big.LITTLE processor architecture equipped with four ARM Cortex-A73 and four Cortex-A53 cores. Every Cortex-A73 core has 64KB L1 instruction caches

Measurement	with L2 (us)	without L2 (us)
LA to STA	98	[88]
LA to TA	123	[120]
LA to SA	150	249
SA to TA	310	353

Table II: Performance Sanctuary Communication, square brackets indicate that deactivating L2 for the SANCTUARY Instance had no effect.

and 64KB L1 data caches. Moreover, all Cortex-A73 cores share a unified L2 with a size of 2MB. The energy-efficient Cortex-A53 cores share a unified L2 cache of 512KB. Besides, every Cortex-A53 core has exclusively access to 32KB L1I and 32KB L1D caches.

A. Microbenchmarks

We evaluated the performance of SANCTUARY by measuring the run time of the individual components and operations of our prototype. We performed the evaluation for the SANCTUARY configurations with both, active and deactivated L2 cache for the SANCTUARY core (other cores are unaffected by this). For an active L2 cache, we consider a weaker attacker model, that is similar to the one of Intel SGX, i.e., side-channel attacks are out of scope; orthogonal approaches like cache partitioning are needed. Furthermore, we assume that the identity-based filtering is also implemented in shared L2.
When not caching the SANCTUARY memory in L2, we can consider a stronger adversary that leverages software side-channel attacks. Square brackets in the results for the configuration without L2 highlight that these measurements are not influenced by the SANCTUARY L2 cache configuration. The shown deviations can be attributed to the complexity of modern processors which causes timing differences between consecutive runs. We used the generic timer available on ARM-based architectures to perform all our measurements. Moreover, we computed the relative standard deviation of our measurements to assess SANCTUARY's stability. The presented results are averaged over 100 runs per configuration. Based on these numbers, we conclude that latency introduced by SANCTUARY is practical in real-world applications.

1) Sanctuary Communication: Table II contains measurements for the different communication channels that exist in the SANCTUARY design. The first two measurements, *LA to STA* and *LA to TA* show how long it takes to perform a call from an LA to a TA or a STA. These measurements are completely independent from a SANCTUARY Instance but can be used to assess the performance of SANCTUARY's communication channels. The time required to perform a call from an LA to its SA with L2 cache is comparable to regular TrustZone communication. Hence, SANCTUARY does not introduce a large communication latency. The higher overhead for the communication between SA and TA is caused by the fact that the context switch is not performed by the SANCTUARY core but is triggered by the corresponding LA. This means the SA first has to communicate with the normal world before it can communicate with the secure world. As mentioned in Section V, the OP-TEE driver could also be included into the SL. Then, the SANCTUARY core could switch directly to the secure world. In this case performance similar to that of a call from LA to TA can be expected. When the L2 cache is deactivated for SANCTUARY memory, the duration of a call from LA to SA increases by a factor of 1.66, however the overall performance is still practical. The relative standard deviation of the communication measurements is low with 28%-34% for the configuration with L2 activated and 20%-32% for the configuration with L2 deactivated.

Measurement	with L2 (ms)	without L2 (ms)
Load Sanctuary binaries	7	[7]
Shut down core	113	[109]
Lock & Verify	13	[12]
Start Sanctuary:	59	311
Early core initialization	37	36
Set up kernel space env.	18	130
Set up user space env.	4	145

Table III: Performance Sanctuary Setup.

2) Sanctuary Setup: The primary difference in running SANCTUARY Instance compared to TAs lies in the setup time needed to isolate a CPU core. The bare execution speed of SAs and TAs is the same as they run on the same hardware. Table III breaks down the single steps performed starting the LA, requesting a SANCTUARY Instance initialization, up to execution of the SA. In the *Load Sanctuary binaries* step, both the SL and SA binaries are loaded in 7ms. In the next step, the Linux hotplug mechanism is used to shut down the core. With L2 cache enabled for this core, this represents the most expensive step of the SANCTUARY setup process with 113ms. Next, the SANCTUARY is locked and verified (cf. Section V). Subsequently, the Zircon kernel is booted (*Start Sanctuary* step). We measured the boot process in three phases. The first phase covers early initialization of the core. In the second phase, the platform components are initialized and the kernel environment is set up. In the last phase, the user space environment is set up, it ends with the execution of the SA. The results show that the boot overhead is higher if the L2 cache is not active. In the second boot step, the boot time increases by a factor of 7, in the third step even by a factor of 36. However, even without using the L2 cache for the SANCTUARY core, the complete SANCTUARY setup can still be performed in around 450ms. If the identity-based filtering feature is implemented in the cache, a setup time around 200ms can be achieved. Further optimizations could be achieved by reducing the SL. The relative standard deviation of the measurements with L2 activated range from 27% to 38%. With deactivated L2 the relative standard deviation values range from 26% to 44%.

3) Sanctuary Teardown: Table IV shows the performance evaluation of the Sanctuary teardown. In the *Sanctuary shutdown* step, the L1 cache is invalidated and the Zircon kernel brought into its original state. In the *Unlock Sanctuary* step, the SANCTUARY memory is zeroed which takes up most of the time. The complete teardown of the SANCTUARY can be

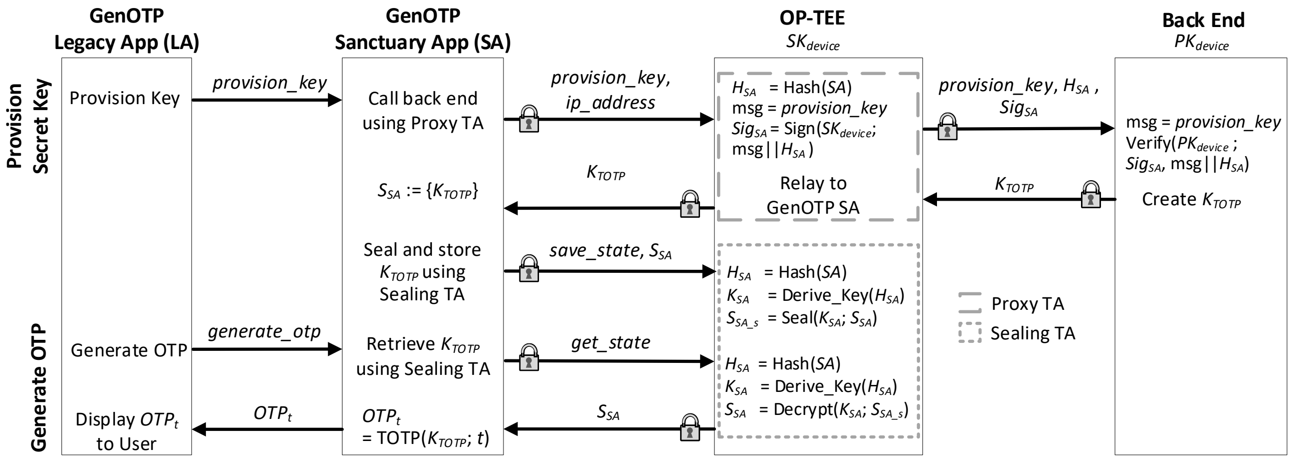

GenOTP Legacy App (LA)
GenOTP Sanctuary App (SA)
OP-TEE SK_{device}
Back End PK_{device}
Provision Secret Key
Generate OTP
Provision Key
provision_key
Call back end using Proxy TA
provision_key, ip_address
H_{SA} = Hash(SA)
msg = provision_key
Sig_{SA} = Sign(SK_{device}; msg||H_{SA})
Relay to GenOTP SA
provision_key, H_{SA}, Sig_{SA}
msg = provision_key
Verify(PK_{device}; Sig_{SA}, msg||H_{SA})
Create K_{TOTP}
K_{TOTP}
S_{SA} := {K_{TOTP}}
K_{TOTP}
Seal and store K_{TOTP} using Sealing TA
save_state, S_{SA}
H_{SA} = Hash(SA)
K_{SA} = Derive_Key(H_{SA})
S_{SA_s} = Seal(K_{SA}; S_{SA})
Proxy TA
Sealing TA
Generate OTP
generate_otp
Retrieve K_{TOTP} using Sealing TA
get_state
H_{SA} = Hash(SA)
K_{SA} = Derive_Key(H_{SA})
S_{SA} = Decrypt(K_{SA}; S_{SA_s})
Display OTP_t to User
OTP_t
OTP_t = TOTP(K_{TOTP}; t)
S_{SA}

Figure 6: GenOTP app protocols for secret key provisioning and OTP generation. A lock symbol indicates TrustZone-protected communication channels.

Measurement	with L2 (ms)	without L2 (ms)
Sanctuary shutdown	1	[1]
Unlock Sanctuary	45	58
Restart core	53	[54]

Table IV: Performance Sanctuary Teardown.

performed very fast in around 100ms with and without L2 cache. For the former case, the relative standard deviation ranges from 15% to 40% and for the later from 14% to 25%. The measurements further emphasize the practicability of SANCTUARY, as setup and teardown induce a total run time overhead of approximately 340ms with L2, respectively approximately 600ms without L2.

B. Use-Case: OTP Generation for Two-Factor Authentication

To illustrate the practicability of SANCTUARY in real-world applications, we implemented a One-time Password (OTP) generator app on top of our prototype, which we call *GenOTP*. The *GenOTP* app, which we will now describe in more detail, consists of an LA and an SA. It can be used to seal a secret key to the SA and restore it at a later point in time to generate a fresh OTP. With SANCTUARY, every service provider can develop a custom app that protects the secret key without the need of an own TA in the TEE.

1) Scenario Description: Two-factor authentication schemes are often used for authenticating users on websites. The first factor, the knowledge factor, is usually represented by an username and a password. The second factor, the possession factor, is represented by a hardware token or a mobile device that creates fresh OTPs. The OTPs are created from a secret key shared between the user's device and the verification server. In a Time-based One-time Password Algorithm (TOTP) [30], the secret key is then used together with a fresh timestamp to generate an OTP. The secret key must be securely stored on the device and the TOTP code protected during execution.

In our scenario, an online retailer wants to offer two-factor authentication for its online shop. We assume that a customer's mobile device contains a TEE and supports SANCTUARY. The online retailer implemented the *GenOTP* app consisting of non-sensitive code in an LA and security-sensitive code in an SA. We further assume that the *GenOTP* app is already installed on the user's device and that the TEE contains an unique asymmetric key pair $(SK_{device}, PK_{device})$, brought onto the device during production. During the installation of *GenOTP*, PK_{device} was sent to the retailer's back end. We assume that *Proxy TA* and *Sealing TA* are present on the device.

2) Provision Secret Key: For generating OTPs on the mobile device, a secret key K_{TOTP} needs to be provisioned to it. The process for receiving the key from the retailer's back end is shown in Figure 6. The customer selects the option to provision a key in the *GenOTP* LA. A SANCTUARY Instance is started and executes the *GenOTP* SA. The SA then hands over the IP address of the server it wants to communicate with and the message it wants to send to the *Proxy TA*. In this case, the message only contains the information $provision\_key$. The *Proxy TA* now calls the STA to get the hash value of the SA binary running in the SANCTUARY Instance and creates a signature Sig_{SA} over the hash value and message using the device unique private key SK_{device}.
After the *Proxy TA* created the signature Sig_{SA}, it sends it to the retailer's back end, together with the hash H_{SA} calculated over the SA binary and the message $provision\_key$. In particular, the signed and thus protected message and H_{SA} is passed to the network stack in the normal world to be forwarded to the server. The involvement of the normal world is omitted in Figure 6 for lucidity as it is only providing non-secure functionalities. The retailer's back end, which has PK_{device}, can now verify if the SA was correctly loaded in a SANCTUARY Instance, since only then a valid signature is created by the *Proxy TA*. If verification succeeds, a secret key K_{TOTP} is created and returned via the *Proxy TA* to the SA.
The SA now needs to store the received key s.t. fresh OTPs can be generated anytime, even without Internet connection. For this, the *Sealing TA* is used. The SA collects all data it wants to seal in a state object S_{SA} and forwards it to the *Sealing TA*.

In our scenario, S_{SA} only contains the secret key K_{TOTP}. In general, any data that needs to be stored persistently can be incorporated into S_{SA}. When the *Sealing TA* receives S_{SA}, it calls the STA to get the SA binary hash. From the hash a symmetric key K_{SA} unique to the SA is derived. K_{SA} is then used to seal S_{SA} to the specific SA, producing the cipher S_{SA_S}. Finally, the data is sealed to the *Sealing TA* using functionality provided by OP-TEE.

3) Generate OTP: When the customer later wants to generate a fresh OTP for login into the retailer's online shop, he selects the OTP generation option from the LA. After a SANCTUARY Instance is started, the SA uses *get_state* of the *Sealing TA* to retrieve its saved data. The *Sealing TA* first restores the sealed SA state S_{SA_S} using the functionality provided by OP-TEE. Next, a hash computed over the SA binary is received from the STA and used to derive the SA unique key K_{SA}. The key is then used to decrypt S_{SA_S} which results in the state object S_{SA}. S_{SA}, which contains the secret key K_{TOTP}, is then returned to the *GenOTP* SA. Finally, the SA runs the TOTP algorithm to compute a fresh OTP_t from the key K_{TOTP} and the current timestamp t. The generated OTP is returned to the LA which displays it to the user. The customer can now use the OTP to perform a two-factor authentication.

4) GenOTP Performance: Besides performing microbenchmarks We also measured performance of the implemented *GenOTP* app. Averaging over 100 runs, we measured the time it takes to perform the *Provison key* and the *Generate OTP* processes shown in Figure 6. The results are listed in Table V. Provisioning a key onto the device takes around 1s, whereas the provisioning time increases by factor 1.3 without L2 cache. Measurements include all steps from SANCTUARY Instance setup, SA signature computation, encrypting and storing the secret key, up to the point where the SANCTUARY Instance is completely teared down. Only the communication and processing delays introduced by the back end are not included. We again split the measurement into multiple phases: (1) SANCTUARY Instance is started and the call to the back end is issued, (2) the secret key is received and processed by the SA, and (3) the secret key is stored, the SANCTUARY Instance teared down and all resources reclaimed by the normal world. The measurement of *generate OTP* is divided into two phases: (1) setup and retrieval of the secret key from the *Sealing TA*, and (2) generation of a fresh OTP and SANCTUARY Instance teardown. Generating a fresh OTP using SANCTUARY takes around half a second. When the L2 cache is deactivated, the process time increases by a factor of 1.6. The relative standard deviation values for the *GenOTP* measurements range from 11% to 21% for the configuration with L2 activated and from 11% to 22% for the configuration with L2 deactivated.

The results show that the SANCTUARY design is indeed practical in real-world scenarios, even without the L2 cache. The setup of the SANCTUARY Instance and the communication with other normal-world and secure-world components is fast enough such that the user experience is not influenced. Moreover, since the SANCTUARY Instance runs on an isolated core, the LOS does not have to be suspended and can run in parallel with the SANCTUARY Instance. This means the delays introduced by the SANCTUARY setup and teardown never result in a frozen UI since the LOS is always fully responsive. Therefore, SANCTUARY fulfills functional requirement 7: *Positive user experience* from Section III-B.

Measurement	with L2 (ms)	without L2 (ms)
Provision key:	884	1174
Setup & Server call	780	1067
Process server result	10	10
Save state & Teardown	94	97
Generate OTP:	365	630
Setup & Retrieve state	266	514
Generate OTP & Teardown	99	116

Table V: GenOTP App Performance.

VIII. RELATED WORK

In this section we compare SANCTUARY against existing TEE implementations in hardware and software.

A. Secure Hardware Architectures

Hardware-based security architectures have been developed by both, academia and industry. Industry solutions like Intel Software Guard Extensions (SGX) [31] and ARM TrustZone [5] are available in commercial off-the-shelf products. Intel SGX [31] provides hardware-enforced code and data isolation, while the TCB consists of the CPU and its microcode only. So-called enclaves run security-sensitive code that can be can be verified via local and remote attestation. However, SGX is tailored to Intel x86 desktop/server chips, and thus not found in embedded (or mobile) devices. For mobile devices, ARM offers a TEE implementation with TrustZone [5]. TrustZone isolates critical code by dividing physical hardware in virtual normal-world and secure-world realms. The secure world runs its own TOS and TAs, but vendors are very strict about which applications may run in the secure world. SANCTUARY overcomes this restriction by only having a minimal and fixed set of functionality in the secure world, while the remaining sensitive code runs in isolated normal-world enclaves.

Sanctum [14] provides protected enclave execution similar to Intel's SGX. Unlike SGX, it extends the open-source RISC-V platform, and provides additional protection mechanisms against side-channel attacks by applying cache partitioning to the last level cache (LLC), while flushing the per-core L1 cache upon enclave exit. SPM [50] and follow-up works like Sancus [41], [42] propose an isolation architecture for low-end embedded systems with a hardware-only TCB. They extend the `openMSP430` CPU architecture with additional CPU instructions for secure provisioning and protected storage, as well as an extended memory access logic with isolation enforcement. TrustLite [34] uses the generalized concept of an Execution-Aware Memory Protection Unit (EAMPU) to enforce program counter based memory access policies stored in tables directly in the SoC and a trusted loader to enable isolated trusted applications on a low-end embedded processor architectures. All these approaches are based on CPU architectures not commonly available in end-user devices, while SANCTUARY is based on the widely used ARM architecture.

B. Secure Software Architectures

Komodo aims to strengthen software isolation between the TrustZone applications in the secure world by using a hardened, formally verified microkernel as the secure-world OS [18]. Komodo replaces deployed microkernels by solutions like MobiCore [7] and hence does not support legacy systems.

Hypervisor-based approaches like vTZ [28], AppSec [46], Terra [21], InkTag [26], TrustVisor [37] or MiniBox [35] provide isolation using virtualization. This has four main disadvantages: (i) their TCB contains a relatively large hypervisor, (ii) they block usage of virtualization for non-security purposes, (iii) they require additional hardware to protect against Direct Memory Access (DMA) attacks, and (iv) they negatively influence the performance of the OS. SANCTUARY does not rely on virtualization and can even be used in combination with a hypervisor. Cho et al. [13] try to mitigate the influence on the OS by activating the hypervisor on-demand. Therefore, the OS is only influenced when sensitive code is executed. In SANCTUARY, the performance of the OS is not influenced when sensitive code is executed in parallel since no hypervisor is running underneath the normal-world OS.

Other approaches try to minimize the normal-world TCB by protecting the non-secure kernel. TZ-RKP [10] and SPROBES [22] both protect the LOS kernel by instrumenting critical functionality to trap into the secure world, where the call is filtered. As demonstrated by the Towelroot exploit [27], such mechanisms can be circumvented. KENALI [48] instead uses data-flow integrity to enforce policies of the LOS kernel's access control system, while SKEE [9] aims to detect attacks against the kernel by providing an isolated execution environment at the kernel's privilege level running a kernel monitor. SANCTUARY does not require the kernel to be trusted to guarantee isolated execution, moreover, SANCTUARY also protects the LOS kernel from potentially malicious SAs.

Flicker [38] and TrustICE [51] provide *temporal isolation* only, i.e., they cannot provide isolation for systems where TEEs execute in parallel with untrusted software. Hence, on todays commonly used multi-core systems the applicability of these approaches is very limited. With temporal isolation, the entire system has to be suspended, i.e., hibernation of the LOS and all applications. Afterwards, the TEE can execute exclusively on the system and only after the TEE has terminated, the normal system can be restored and continue execution. Flicker uses Intel's Trusted Execution Technology (TXT) to reset the system at runtime to a trusted execution state. TrustICE is conceptually similar to Flicker: it uses the secure world, rather than TXT, to reset the normal world to a trusted state. In TrustICE, TA binaries are stored in TrustZone memory. When a TEE is started, the LOS is suspended and the binaries are copied to normal-world memory for execution. After the TEE finished execution, the LOS has to be restored by the secure world. During execution, TrustICE provides only one-way isolation and executes in kernel-mode, this means that malicious TAs can manipulate normal-world software, e.g., compromise the LOS. SANCTUARY, in contrast, does provide *spacial isolation*, which enables the *parallel* execution of untrusted code with one or multiple TEE instances. Furthermore, SANCTUARY offers hardware-enforced two-way isolation and restricts SAs to user-mode execution. Hence, SANCTUARY protects systems from malicious SAs, which is highly relevant for practical deployment.

IX. Conclusion

We presented SANCTUARY, our novel security architecture for extending the TrustZone software ecosystem with user-space enclaves. SANCTUARY provides hardware-enforced two-way isolation obviating the need to trust or vet the code of SAs, as malicious SAs cannot have more power than normal user-space applications.

SANCTUARY is based on the bus master identity filtering introduced with ARM's latest memory controller design and allows the parallel isolation of individual CPU cores for executing security-sensitive code, i.e., SANCTUARY does not affect the user experience negatively. Furthermore, our performance evaluations for our proof-of-concept implementation shows low latencies for typical use cases, all of which makes SANCTUARY highly practical.

Acknowledgments

This work was co-funded by the DFG (projects P3 and S2 within CRC 1119 CROSSING, and HWSec), by the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] "OP-TEE," https://www.op-tee.org/.

[2] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[3] ARM Limited, "ARM Cortex-A Series Programmer's Guide for ARMv8-A," http://infocenter.arm.com/help/topic/com.arm.doc.den0024a/DEN0024A_v8_architecture_PG.pdf.

[4] ——, "GlobalPlatform TEE & ARM TrustZone technology: Building security into your platform," https://pdfs.semanticscholar.org/presentation/7b94/63d58a2d4ec9724c5933419be6f08754ce86.pdf.

[5] ——, "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.

[6] ——, "CoreLink TrustZone Address Space Controller TZC-380," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0431c/DDI0431C_tzasc_tzc380_r0p1_trm.pdf, 2010.

[7] ——, "Giesecke & Devrient and ARM Protect Mobile Applications From Data Theft," https://www.arm.com/about/newsroom/26718.php, 2010.

[8] ——, "ARM CoreLink TZC-400 TrustZone Address Space Controller," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0504c/DDI0504C_tzc400_r0p1_trm.pdf, 2013.

[9] A. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "Skee: A lightweight secure kernel-level execution environment for arm," in *23rd Annual Network and Distributed System Security Symposium*, ser. NDSS, 2016.

[10] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS, 2014.

[11] S. S. Bhat, "Interaction of suspend code (s3) with the cpu hotplug infrastructure," https://www.kernel.org/doc/Documentation/power/suspend-and-cpuhotplug.txt, 2014.

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

[13] 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 Annual Technical Conference (USENIX ATC)*, 2016.

[14] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation." in *USENIX Security Symposium*, 2016.

[15] Dan Rosenberg, "Reflections on trusting trustzone," https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf, 2014.

[16] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi, "Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM," in *8th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS, 2013.

[17] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security & Privacy*, 2014.

[18] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP, 2017.

[19] Gal Beniamini, "Qsee privilege escalation vulnerabilitiy," http://bits-please.blogspot.de/2015/08/full-trustzone-exploit-for-msm8974.html, 2015.

[20] ——, "Qsee privilege escalation vulnerabilitiy," http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, 2016.

[21] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP, 2003.

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

[23] Global Platform, "Tee management framework (version 1.0)," https://www.globalplatform.org/specificationform.asp?fid=7866, 2016.

[24] Google, "Zircon micro kernel," https://fuchsia.googlesource.com/zircon.

[25] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[26] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS, 2013.

[27] G. Hotz, "Towelroot android root exploit," https://towelroot.com/, 2014.

[28] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[29] W. Huang, V. Rudchenko, H. Shuang, Z. Huang, and D. Lie, "PearlTEE: Supporting Untrusted Applications in TrustZone," 2018.

[30] IETF, "Totp: Time-based one-time password algorithm - rfc6238," https://tools.ietf.org/html/rfc6238, 2011.

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

[32] C. Josh Thomas, Nathan Keltner, "Reflections on trusting trustzone," https://pacsec.jp/psj14/PSJ2014_Josh_PacSec2014-v1.pdf, 2014.

[33] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*.

[34] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 10.

[35] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[36] J. 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.

[37] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[38] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *ACM SIGOPS Operating Systems Review*, 2008.

[39] 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 *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[40] M. Meeker, "Internet trends 2015," *Glokalde*, vol. 1, no. 3, 2015.

[41] 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 *22nd USENIX Security symposium*, 2013.

[42] J. Noorman, J. V. 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," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, p. 7, 2017.

[43] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *RSA Conference*, 2006.

[44] Project Zero, "Lifting the hyper visor," https://googleprojectzero.blogspot.de/2017/02/lifting-hyper-visor-bypassing-samsungs.html, 2017.

[45] ——, "Trust issues: Exploiting trustzone tees," https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017.

[46] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi, "Appsec: A safe execution environment for security sensitive applications," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE, 2015.

[47] D. Shen, "Exploiting trustzone on android," https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf, 2015.

[48] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in *23rd Annual Network and Distributed System Security Symposium*, ser. NDSS, 2016.

[49] N. Stephens, "Behind the pwn of a trustzone," https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose, 2016.

[50] R. Strackx, F. Piessens, and B. Preneel, "Efficient isolation of trusted subsystems in embedded systems," in *Security and Privacy in Communication Networks*, 2010.

[51] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[52] Tencent, "Defeating samsung knox with zero privilege," https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege-wp.pdf, 2017.

[53] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security Symposium*, 2014.

B

Offline Model Guard: Secure and Private ML on Mobile Devices (DATE'20)

[12] 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 *Conference on Design, Automation and Test in Europe (DATE)*, pages 460-465, 2020. CORE Rank B. Appendix B.

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.

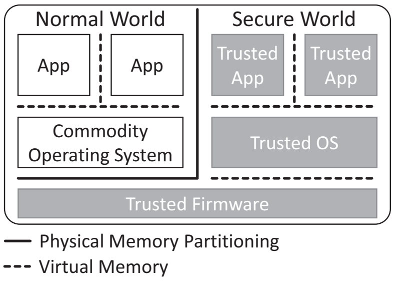


Fig. 1: ARM TrustZone architecture overview.

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:

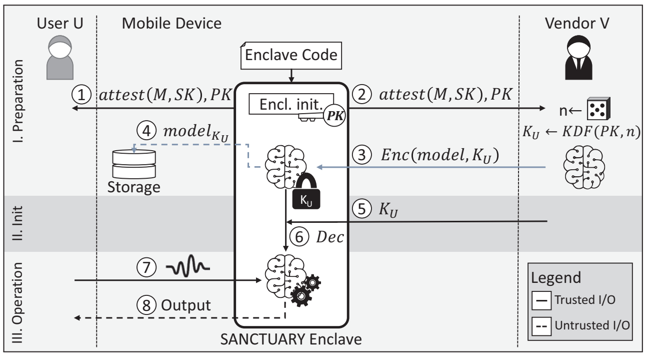


Fig. 2: OMG overview. Once the encrypted model is stored locally, steps in gray are optional until a model update.

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, "Prεεch: 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.

C

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

[9] 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 *USENIX Security Symposium*, 2021. CORE Rank A*. Appendix C.

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

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

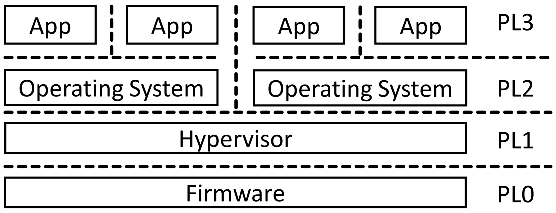


Figure 1: Software privilege levels (PL): user space, kernel space & dedicated levels for hypervisor & firmware.

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

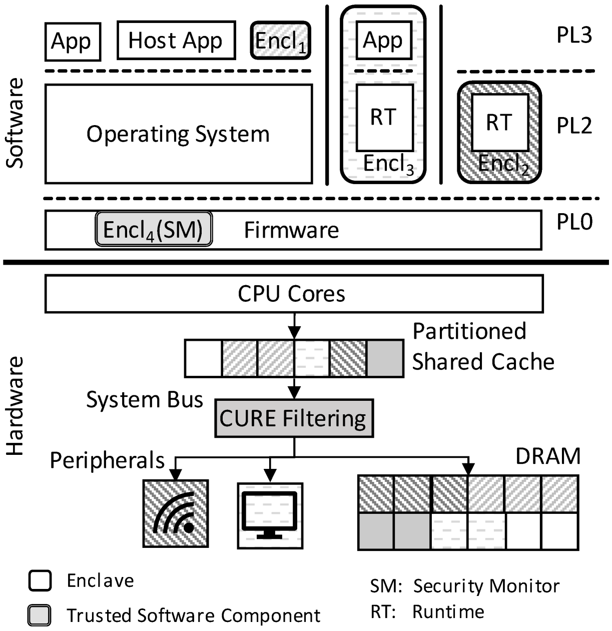


Figure 2: CURE privilege levels and enclave types, namely, user-space enclaves (Encl$_1$), kernel-space enclaves (Encl$_2$, Encl$_3$) and sub-space enclaves (Encl$_4$).

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

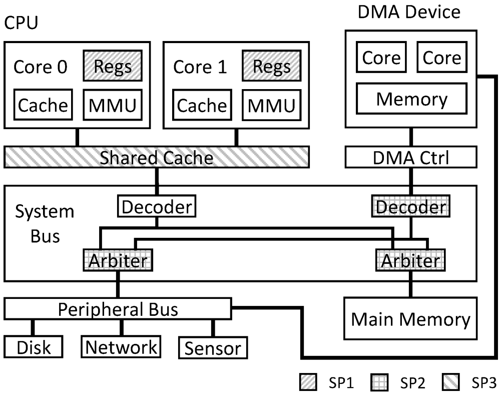


Figure 3: CURE Security Primitives (SPs), added at core register files (SP1), system bus (SP2) and shared cache (SP3).

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].

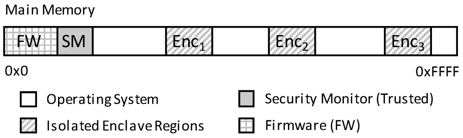


Figure 4: Physical memory layout of our CURE prototype.

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.

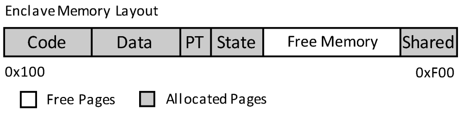


Figure 5: CURE enclave memory layout consisting of the code & data pages, page tables (PT), the enclave state (State) and the shared memory (Shared).

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 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.

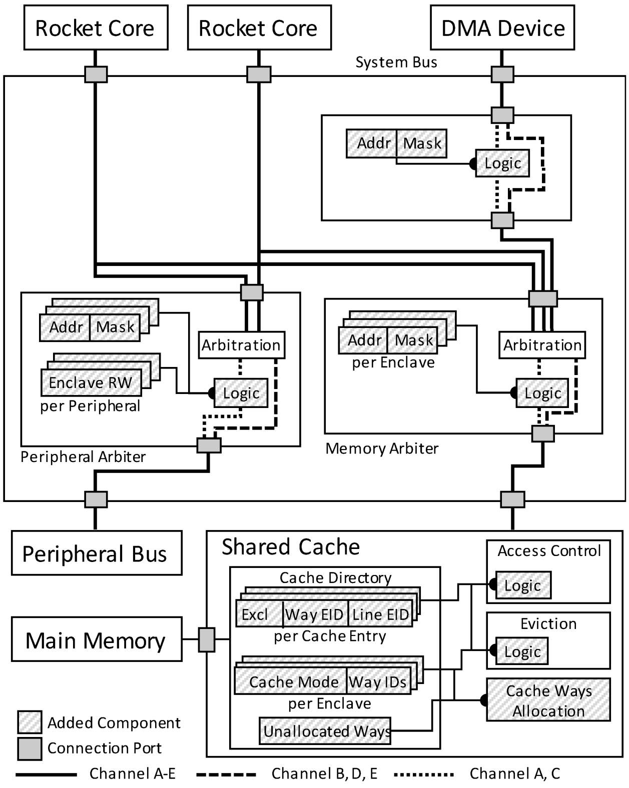


Figure 6: CURE prototype implementation using Rocket Chip.

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

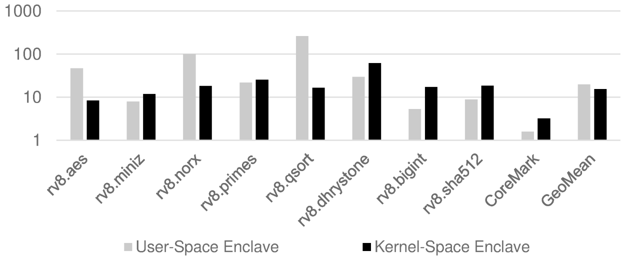


Figure 7: CURE performance overhead (in percent) on macro benchmarks `rv8` and `CoreMark` relative to a normal process.

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

		Enclave Type					
Name	Extensions	User-Space	Kernel-Space	Sub-Space	Dynamic Cache Side-Channel Resilience	Controlled Side-Channel Resilience	Enclave-to-Peripheral Binding
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.

D

Distributed Memory Guard: Enabling Secure Enclave Computing in NoC-based Architectures (DAC'21)

[61] Ghada Dessouky, Mihailo Isakov, Michel A. Kinsy, Pouya Mahmoody, Miguel Mark, Ahmad-Reza Sadeghi, Emmanuel Stapf, and Shaza Zeitouni. **Distributed Memory Guard: Enabling Secure Enclave Computing in NoC-based Architectures**. In *ACM Design Automation Conference (DAC)*, pages 985-990, 2021. CORE Rank A. Appendix D.

Distributed Memory Guard: Enabling Secure Enclave Computing in NoC-based Architectures

Ghada Dessouky[*], Mihailo Isakov[†], Michel A. Kinsy[†], Pouya Mahmoody[*], Miguel Mark[†],
Ahmad-Reza Sadeghi[*], Emmanuel Stapf[*], Shaza Zeitouni[*]
[*]*Technical University of Darmstadt*, [†]*Texas A&M University*
[*]{ghada.dessouky, pouya.mahmoody, ahmad.sadeghi, emmanuel.stapf, shaza.zeitouni}@trust.tu-darmstadt.de
[†]{mihailo, mkinsy, mmark}@tamu.edu

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586222

***Abstract*—Emerging applications, like cloud services, are demanding more computational power, while also giving rise to various security and privacy challenges. Current multi-/many-core chip designs boost performance by using Networks-on-Chip (NoC) based architectures. Although NoC-based architectures significantly improve communication concurrency, they have thus far lack adequate security mechanisms such as enforceable process isolation. On the other hand, new security-aware architectures that protect applications and sensitive services in isolated execution environments, i.e., enclaves, have not been extended to provide comprehensive protection for NoC platforms. These enclave-based architectures (i) lack secure enclave-device interaction, (ii) cannot include unmodifiable third-party IP, or (iii) provide flexible enclave memory management.**

To address these design challenges, we introduce a new hardware security primitive, the Distributed Memory Guard, and design the first security architecture that protects sensitive services in NoC-based enclaves. We provide evaluation of this reference architecture and highlight the fact that one can design a scalable (i.e., NoC-based) and secure (i.e., enclave-based) architecture with minimal hardware complexity and system performance overhead.

***Index Terms*—Secure Processor Design, Enclave, Network-on-Chip, Memory Protection**

I. Introduction

Today, around 60% of the global internet traffic is caused by cloud streaming services [1]. In order to cope with the high demand for computational power, chip vendors incorporate increasing numbers of processors on a single chip. When connecting a large number of heterogeneous processing units, e.g., CPUs, GPUs or specialized hardware accelerators, traditional bus architectures typically deployed on System-on-Chips (SoC) become impractical. Thus, new chip designs use scalable bus architectures to connect processing units, memory and devices. These Network-on-Chip (NoC) designs are deployed by processor manufacturers such as Intel and AMD, as well as by e.g., companies licensing the ARM Neoverse designs.

Many cloud services process sensitive data, such as, privacy-sensitive user data, intellectual property of companies, or even sensitive governmental data. Therefore, adequate security measures are required to protect the sensitive data and computations. One prominent approach is to use *enclave security architectures*. In enclave architectures, hardware security mechanisms, either controlled by microcode or a small software component, are used to isolate sensitive applications in secure execution contexts, called *enclaves*. In contrast to sandboxing approaches (e.g., [2], [3]), enclave architectures provide strong isolation capabilities that even protect from compromised system software, e.g., the operating system kernel or hypervisor. The high demand for such security solutions led to the development of AMD's Secure Encrypted Virtualization (SEV) [4] and Intel's Software Guard Extensions (SGX) [5]. On RISC-V and ARM, several proposed solutions exist, most notably, Sanctum [6], Sanctuary [7], Keystone [8] and CURE [9].

However, currently deployed enclave security architectures cannot protect real-world sensitive services in a comprehensive way since they either i) do not enable a secure interaction between enclaves and devices (secure I/O) which is required in use cases such as *Machine Learning as a Service* where computations are offloaded to ML accelerators, ii) do not support the integration of unmodifiable hardware IP blocks from third-party vendors (which is common practice on ARM and RISC-V systems) into the chip design since they are not compatible with the enclave architecture, or iii) do not provide a flexible management of enclave memory comparable to what operating systems offer, which prevents supporting a larger number of enclaves in parallel.

Recent enclave security architectures [8], [9] tackle the aforementioned challenges i) and ii) by moving their hardware security mechanisms out of the processing units, in contrast to other solutions [4], [5], [6]. This enables enclaves to execute privileged software and to isolate complete processing units from the rest of the system. However, both approaches only provide modest management capabilities for the enclave memory and thus, require allocating contiguous memory regions at runtime. As we show in Section II, this is very difficult since physical memory quickly becomes heavily fragmented. Reserving a large memory region at boot time is also not practical since this requires prior knowledge about the enclave memory requirements. Furthermore, for multi-enclave systems, even the reserved region of memory will become fragmented. In summary, the existing approaches cannot be efficiently used to protect unmodified memory-intensive and security-sensitive services.

Our goal and contributions. In this paper, we present a novel hardware security component, called the Distributed Memory Guard (DMG), designed to bring enclave computing capabilities to NoC-based platforms. The DMG is placed between every processing unit and router in the NoC and performs access control on memory transactions using a set of rules maintained by a trusted software component. Since the rules apply to physical memory addresses, privileged software execution in enclaves (and thus secure I/O) can be supported. In order to tackle high memory fragmentation, we design rules that can specify non-contiguous memory regions. When allocating memory for an enclave, this flexible rule design is used to assign a subset of the free memory pages to the enclave. By having the flexibility to allocate enclave memory within the fragmented free memory instead of clearing out large contiguous regions, we can i) improve enclave performance since we decrease the added latency of enclave memory allocations, ii) use the same physical memory zone for both enclaves and non-sensitive applications by providing a strong separation in physical memory. As a result, when using the DMG as the underlying security mechanism, enclave architectures can be designed, which, in contrast to related work, facilitate the protection of service workloads in strongly-isolated enclaves, even in the presence of a highly fragmented memory space.

In summary, our contributions are as following:

- We analyze the cloud service benchmark CloudSuite [10], its sources of fragmentation, and show that memory fragmentation is essentially unavoidable on long-running systems.
- We design and implement the Distributed Memory Guard (DMG), a novel hardware security component which performs NoC-level access control on outgoing memory requests. We evaluate the DMG's novel rule design and compare against the RISC-V Physical Memory Protection (PMP) unit.
- We design a novel secure enclave architecture for NoC-based computing platforms based on the DMG security component.

II. The Problem of Physical Memory Fragmentation

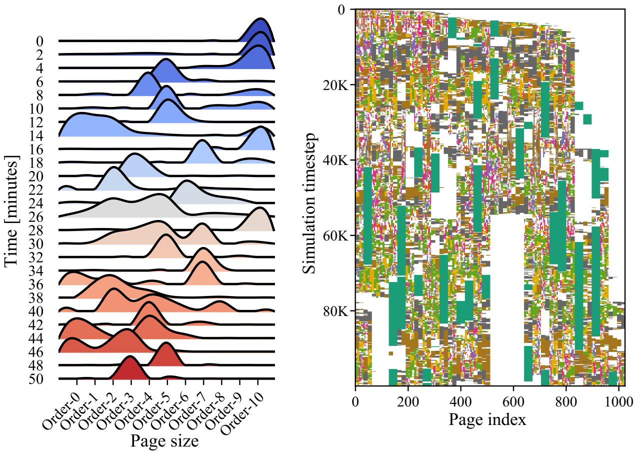
Physical memory fragmentation occurs when free memory is partitioned into many small separate sections, and the system is unable to satisfy large contiguous allocations, even though a large percentage of memory is in fact unused. We next describe our investigation of physical memory fragmentation on systems which run realistic cloud services, and highlight why this is an often overlooked but non-trivial problem for enclave security architectures.

Memory allocation fundamentals. Typically, main memory is split into several zones dedicated to e.g., the kernel and user-space applications. Each zone has an allocator that maintains a logical tree-like data structure that maintains the state (used / free) of each page of physical memory. In Linux, this structure is the buddy allocator [11], which maintains 11 linked lists of pages of different sizes, with the order-0 list containing pointers to 4KiB pages, while order-n lists contain pointers to pages of size $2^n \times$4KiB. When an allocation of 4KiB is requested by a process, the buddy allocator checks for available order-0 pages. If not found, it will progressively check the larger order lists. If an order-1 page is available, it will split the order-1 page into two order-0 pages, and allocate one of them to the process, while keeping the second page for later use. The largest available order that the buddy allocator keeps is order-10, corresponding to 4MiB of contiguous physical memory, hence why the Linux `kmalloc` can allocate at most 4MiB of memory.

Physical memory fragmentation. To investigate how memory fragmentation impacts enclave memory management, we run an experiment where we monitor the memory fragmentation of a freshly restarted Linux server over the course of several hours. The system has 2 Intel Xeon E5645 processors and 32GiB of RAM and runs Ubuntu 20.04 and the 5.4.0-54 kernel. We run realistic cloud service workloads on the system using the CloudSuite [10] benchmarks that represent real-world cloud services. We keep the server at low to medium load, and make sure that the system never utilizes more than 70% of available RAM. As our experiments show, fragmentation will still occur even on moderately utilized systems.

In Figure 1a, we show a series of distributions of available pages collected by periodically reading `/proc/pagetypeinfo`. Histograms were collected at 2-minute intervals, where each histogram shows the fragmentation of physical memory. The total area under the histogram represents all free memory, and each bar shows what percentage of free memory resides in pages of the specified size. We observe that after booting (blue histograms at top of figure), most of free memory resides in order-10 pages, which correspond to 4MiB-large contiguous memory regions. As the system gets warmed up and memory gets more fragmented, it runs out of order-10 and order-9 pages, despite still having plenty of available free memory.

We also evaluate how killing applications affects fragmentation, where at the 28-minute mark, we *synchronously* kill and restart all of the benchmark applications. We evaluate whether enough memory will be reclaimed by the OS such that the buddy allocator groups together the smaller fragmented pages and builds a large number of order-10 pages. While this does happen, the system is still more fragmented when compared to the 'cold' system. This is because other running applications keep allocating memory in parallel which is scattered throughout the fragmented memory space. This illustrates that even best-case scenarios are insufficient to revert physical memory fragmentation, and thus physical memory will always remain extremely fragmented on any long-running system.



(a) The numbers and sizes of free memory regions at 2-minute intervals. Each histogram shows the number of order-n pages the OS buddy allocator's possesses.

(b) The state of physical memory at different time periods, with free and allocated pages shown in white and red, respectively. Each cell represents the status of a single order-0 page at the specific time.

Fig. 1: Progressive fragmentation of a system (left) and the physical memory organization of a system over time (right).

Insights into memory fragmentation. To get more insight into memory fragmentation, we collect memory access traces of several applications, as seen by the kernel, to reconstruct the full state of the buddy allocator when scheduling a set of different applications. We explore CloudSuite benchmarks as well as a set of simple handcrafted applications that exhibit similar fragmentation, but with minimal program complexity. Different handcrafted applications have different behaviors, where some applications can only free the most recently allocated memory (like a stack) or can free at random any previously allocated memory (like a heap). We schedule these applications and feed their allocations to our buddy allocator simulator, from which we reconstruct the state of physical memory at every timestep. In Figure 1b, we show an example configuration of 8 crafted applications allocating and freeing memory using a buddy allocator. Every cell of the figure represents the occupancy of a 4KiB page at a given timestep, where a white cell represents that the page is free, and other colors each correspond to a different application. We can see that initially, memory is mostly free and not fragmented, but as applications progress memory quickly gets fragmented, despite never being fully utilized.

Through experimentation with our simulator we conclude that: i) heap-like programs cause significantly more fragmentation than stack-like programs, since freeing memory from 'the middle' prevents the buddy allocator from grouping the free memory with subsequent regions, and ii) if the running programs only allocate pages of one size (e.g., order-3 pages), fragmentation will be kept low. Only through interaction of programs that have significantly different allocation sizes will memory become fragmented.

Fragmentation in enclave computing. The problem of memory fragmentation is not new. On commodity systems, memory fragmentation is countered by using a Memory Management Unit (MMU). This introduces a level of abstraction through virtual memory, thus hiding the fragmented physical memory space and presenting a contiguous virtual memory space to user-space applications. Therefore, several enclave architectures [4], [5], [6] perform access control at the MMU-level *within* the processing units, i.e. at the virtual-to-physical memory address translation. However, these architectures cannot integrate unmodifiable IP blocks and accelerators whose processors are not also similarly modified internally with compatible access control mechanisms. Moreover, since physical memory addresses are not protected, privileged software components, such as device drivers, cannot be executed within these enclaves. On the contrary, more recent architectures [9], [8] overcome these limitations by integrating their hardware security mechanisms *out* of the processing units and by applying access control mechanisms on physical memory addresses. However, this requires that enclaves allocate a limited number of large contiguous physical memory regions. As we demonstrate above, in practice, memory fragmentation makes it almost impossible for multiple enclaves to allocate large contiguous memory regions along with other running applications. While compaction (a process where the OS relocates allocated pages and updates page table entries [12]) may help mitigate memory fragmentation, this is not practical for memory-intensive applications since it requires copying large amounts of memory which incurs large performance overheads.

We consider this to be one of the most significant challenges of such enclave architectures that has not been sufficiently and practically addressed. Moreover, it is an even more prominent challenge when scaling enclave computing to heterogeneous NoC-based architectures, thus motivating our work.

III. System Assumptions & Adversarial Model

In the following section, we describe our system assumptions and the underlying adversary model.

System assumptions. The Distributed Memory Guard (DMG) was designed for heterogeneous computing platforms which connect a multitude of computing nodes on a single chip over a mesh network, thereby forming a Network-on-Chip (NoC) infrastructure. As illustrated in Figure 2, a computing node can be an on-chip processing unit, e.g., a CPU or specialized processor or accelerator, or a controller which connects to off-chip memory, devices or external processing units, e.g., a graphics card. All computing nodes are connected to the network infrastructure (consisting of routers) over network interface (NI) components. We assume that the vendor of the computing platform combines IP blocks from various third-party vendors on the NoC chip, where some IP blocks cannot be modified by the platform vendor (e.g., the ML processor in Figure 2). However, we assume that at least one of the processing units can be modified and adequate security mechanisms implemented such that a small inherently trusted software component can be provided (comparable to the trusted firmware on ARM) which has exclusive access to the network infrastructure (routers and NIs) over Memory Mapped IO registers. Furthermore, we assume that a memory encryption engine [13], which is out of this paper's scope, can be deployed on the network interface connecting to a memory controller in order to transparently encrypt and authenticate all data leaving the NoC.

Adversary model. The main goal of our assumed adversary is to get access to data processed by a security-sensitive service running on the computing platform. We assume an adversary that is able to compromise the complete software stack on the third-party IP blocks since they do not provide adequate security mechanisms. On the vendor IP blocks, the adversary is able to compromise all system software, e.g., the operating system kernel or hypervisor, except the trusted software component. As a result, our assumed strong adversary is able to freely start malicious privileged processes or to issue memory transactions on the physical address level. Aligned with related work on enclave computing [7], [9], [8], [6], we do not consider physical attacks on the on-chip components, such as, physical side-channel attacks [14] or fault injection attacks [15]. Moreover, we consider all hardware of the platform to be correct and thus, exclude attacks that exploit hardware flaws [16]. We do not consider Denial-of-Service attacks, whether performed on the computing nodes or the network infrastructure since such attacks do not leak sensitive data. Moreover, we stress that side-channel attacks on the bus architecture [17] as well as on cache structures inside of a processing unit [18], [19] are *not* in scope of this paper. These problems have been addressed in several works such as [20], [21].

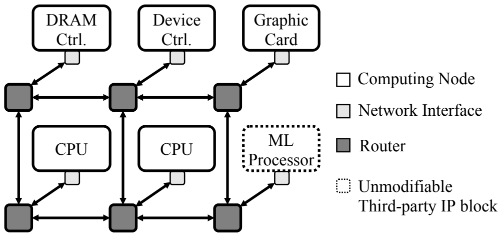


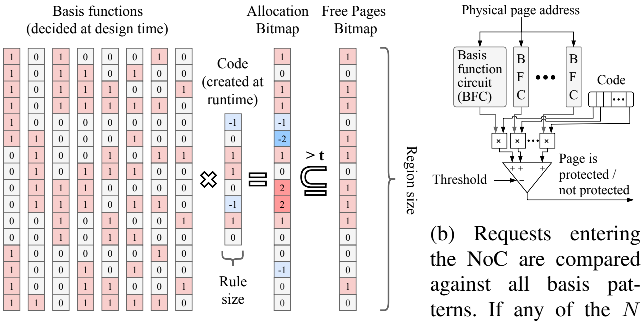
Fig. 2: Heterogeneous computing platform connecting multiple computing nodes over network interfaces with a mesh network of routers.

IV. The Distributed Memory Guard

Protecting physical memory on NoC platforms requires rethinking where protection boundaries should be drawn since not all processing units on the NoC can be extended with adequate security mechanisms. Thus, we propose that the NoC should incorporate the responsibility of preventing illegal memory accesses. In this work, we present a design where the NoC fabric is equipped with a new security primitive, called the Distributed Memory Guard (DMG), which enforces physical memory protection *on every node in the network*. This secure NoC validates communication entering and leaving the network, allowing designers to incorporate unmodifiable IP blocks without sacrificing security. In the remainder of this section, we discuss why the existing access control mechanisms are not flexible enough and present our design of the pattern-based DMG rules. In sections V-B and V-C we provide details on the DMG architecture and how the trusted computing base configures its rules.

A. The Challenge of NoC-level Access Control

One of the main challenges when performing access control on the NoC level is providing a flexible mechanism to assign physical memory regions to enclaves in the presence of memory fragmentation. As we show in Figure 1b, even if a warm system has plenty of available memory, when an enclave requests an allocation, the OS cannot provide it with a contiguous region of memory. Approaches that rely on the RISC-V Physical Memory Protection (PMP) provide a control mechanism where one PMP rule specifies one contiguous memory region. When used while assigning memory regions from fragmented physical memory to enclaves, multiple PMP rules need to be created. This does not scale, and our experiments show that even



(a) Creation of codes that define allocation bitmaps which ideally are a subset of the free pages bitmap.

(b) Requests entering the NoC are compared against all basis patterns. If any of the N codes resonates with the address, the request is matched to the rule.

Fig. 3: DMG rule architecture (a) high-level and (b) hardware level.

small programs require thousands of PMP rules, while the RISC-V standard prescribes a maximum of 64 PMP registers.

Defining large contiguous enclave memory *zones* at boot time is also not practical. While isolating enclave memory into a separate zone can prevent user-space programs from fragmenting the enclave zone, this approach only works when the system has *one* active enclave. With multiple enclaves sharing a memory zone, fragmentation once again becomes an issue. Moreover, choosing the right size for the enclave memory zone would require knowledge about the number of enclaves that will run in parallel on the system and their memory consumption, which is unrealistic in practice. Therefore, in this work, we propose a new type of flexible access rules that do not necessarily specify contiguous memory regions, but can 'fit in' and protect a subset of free pages in a heavily fragmented memory space.

B. The Pattern-based DMG Rules

To protect a set of pages, memory protection rules are required to specify that set. RISC-V PMP rules protect memory by specifying the start of a protected memory region and its power-of-two length. We seek, however, a method for specifying rules that is more flexible and allows 'gaps', such that it can accommodate the fragmented physical memory space. At the finest granularity, bitmaps can be used to indicate protected / unprotected pages. Although this provides ultimate flexibility, it scales linearly with the number of pages we want to protect. Instead, we seek a trade-off approach that may be less flexible but more memory-friendly.

We draw inspiration from Walsh-Hadamard Transforms (WHT), which decompose a discrete sequence into a superposition of basis functions called Walsh functions. The WHT can decompose a 2^n bit long binary sequence into a weighted sum of 2^n binary basis vectors, and thus can be used to specify every free page in a region of memory. However, the WHT still do not provide a simpler representation compared to bitmaps. Our insight is that we do not need to represent and allocate *every* free page in a given region. Instead, we select to represent only a subset of the free pages, thus allowing us to allocate memory from a fragmented memory space while still ensuring rule sizes scale sub-linearly with the number of pages.

In Figure 3a, we show the high-level idea of our DMG rule architecture. At design time, we choose a number of basis functions (matrix on the left). Then at run time, when an enclave wants to allocate memory, the trusted software component (being the only component with privileges to configure these rules) searches for a code (column vector in the middle) that can specify a subset of available free pages, while including the least amount of already allocated pages. Any already allocated page that the code specifies

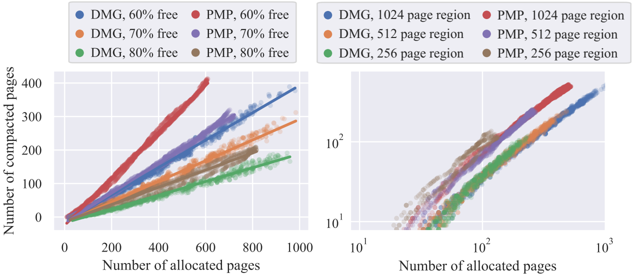


Fig. 4: Number of allocations and compactions DMG and PMP rules can achieve when allocating memory from a uniformly sparse array of pages (left) and from different-sized regions of memory, with memory 50% utilized (right).

will need to be compacted. The set of pages that the code will allocate is the matrix-vector product of the basis function matrix and the code, where all elements larger than a code-specific threshold (second from right column vector) are allocated. The combination of code and threshold represent a rule which is stored in the DMG's rule cache (Figure 5). In Figure 3b, we show how existing rules are enforced in hardware. When a memory request is entering the NoC, it passes through the DMG. The DMG contains a separate circuit for each basis function, and these circuits return a `0` or `1`, depending on whether they 'match' the requested address. The DMG then performs an inner product between the code and the circuit outputs, and if the sum is greater than the code-specific threshold, then the address belongs to the rule and the memory request permitted.

The optimal choice of basis functions (in terms of flexibility in specifying memory regions and hardware complexity) and the algorithm for finding efficient codes are a topic for future work. In this paper, we focus on evaluating whether such a memory protection scheme is feasible. Thus, we evaluate uniformly random basis functions, because otherwise the space of all possible binary functions is very large. We also hypothesize that basis functions may work well because large-enough random vectors are orthogonal, which should increase flexibility.

In Figure 4, we show a set of experiments where we create a uniformly random bitmap of free and used pages, and try to allocate the largest amount of free pages possible while also compacting the least amount of used pages. To find these rules, we use a hill climbing algorithm that iteratively modifies the code, optimizing for the two objectives. We report only the Pareto optimal configurations of allocated / compacted pages that the algorithm arrives at for the DMG and PMP rules. As we can see from Figure 4 (left), while DMG rules are unable to find configurations that do not overlap used pages, they require significantly less compaction, especially when memory is more utilized and less free pages are available. In Figure 4 (right), we see that by looking at larger regions (e.g., 1024 consecutive pages of physical memory) we are able to find larger rules that match these regions, albeit with a linear increase in the number of pages that need to be compacted.

V. Enclave Computing in NoC-based Architectures

To enable secure enclaves on NoCs, we modify the NoC interfaces by adding the DMG modules, and build a security-enhanced core that in cooperation with the DMG can protect the secure execution of enclaves. Next, we explain the architecture of both components.

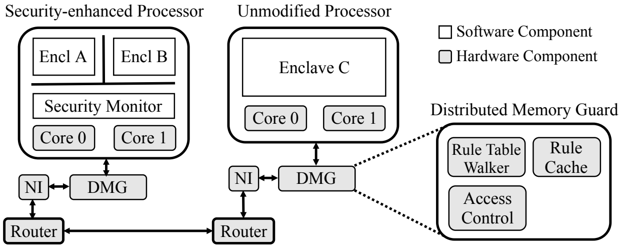


Fig. 5: High-level overview of the DMG and the enclave architecture based on it which provides the enclaves A and B on the security-enhanced processor and enclave C on the unmodified processor.

A. *Architectural Overview*

The location of the DMG on a NoC platform and its basic logic blocks are shown in Figure 5. The main purpose of the DMG, which is placed between each computing node and its corresponding network interface (NI), is to control the node access to the network using rules that specify a set of allowed memory regions. Using the DMG as the underlying security primitive, an enclave architecture can be setup. For this, aligned with our system assumptions detailed in Section III, at least one of the cores must be extended with security mechanisms (we give more details in Section V-D) that allow to strongly isolate a trusted software component from the rest of the system software. We call this trusted component the Security Monitor (SM) and the extended cores *security-enhanced*. In contrast to related work [4], [5], [6], the enclave architecture also supports the integration of unmodified third-party cores in the same NoC since the security primitive is moved out of the processor logic.

B. DMG *Security Monitor (SM) and architecture*

The SM represents the software Trusted Computing Base of the system and is the only component that can configure the DMG. At boot time, the SM performs a sequence of steps. Before loading the OS, the SM creates a rule table in memory, which is initially unprotected. The location of the rule table is decided at design time and is inserted into the device tree and the DMG RTL. Then, the SM creates the first rule that protects the contents of the rule table, such that no other process can create, modify or delete DMG rules.

The SM creates enclaves on the system by generating DMG rules, which assign memory regions exclusively to execution environments. For the unmodified processor shown in Figure 5, the SM creates a single coarse-grained enclave (Enclave C), which comprises the software running on the processor. On the security-enhanced processor, which includes the security extensions (Section V-D), the SM can create fine-grained enclaves which can comprise all software pinned to one processor core (comparable to enclaves in Sanctuary [7] or Keystone [8]), shown as Enclave A & B in Figure 5, or enclaves comprising a single process (comparable to enclave in SGX [5] or Sanctum [6], not shown in Figure 5 for the sake of clarity). Whenever an enclave is created by the SM, a unique enclave ID (`EID`) is assigned to the enclave. `EID` is sent as part of all enclave memory requests. Moreover, the SM creates the DMG rules to specify the memory regions the enclave is allowed to use. Access control is enforced by the DMGs by matching rules with the physical address and the `EID` specified in the memory requests received by a DMG. When one of the last used rules (stored in the rule cache) is matched (rule hit), the request is permitted to enter the network. If no matching rule is found in the cache (rule miss), the rule walker logic searches for a fitting rule in the rule table stored in SM memory. If again no matching rule is found, the request is rejected and an interrupt is triggered. This interrupt is handled by the SM, which executes a predefined access violation policy, e.g., killing the violating process.

C. DMG *architecture*

We show the architecture of a security-enhanced core, it's cache, and the NoC interface in Figure 6. All of the security-related modifications we implement on a baseline design are shown in blue.

The first modification of the baseline design is the addition of the DMG module, which sits between the L2 cache and the network interface. The DMG consists of an access controller, a rule table walker, and a rule cache. At any time, the cache feeds N most recently matched rules into the access controller. Each rule consists of the code, offset, threshold, rule EID, and a valid bit. The access controller (or DMG rule checker in general) evaluates whether outgoing memory requests match any of the cached rules, and if so, permits these requests to enter the network interface. The rule table walker is triggered if a match is not found. It is equipped with a CSR register that specifies where the complete rule table is stored in SM memory. Overall, the DMG is capable of preventing unauthorized requests from entering the NoC. However, there is still the problem of securing (possibly private) memory stored in the caches.

The DMG is designed to allow low-latency evaluation of rules that are present in the rule cache. The DMG critical path contains a basis function circuit and a threshold function, which is implemented as a single cycle operation. When a rule miss happens (i.e., a request does not find a match in the rule cache), the request waits until the rule table walker walks the full rule table. If a request is found, it is placed in the rule cache, otherwise an interrupt is raised.

D. *Security-enhanced Processors*

To prevent a malicious OS or process from avoiding the DMG and reading cache memory, similarly to SGX [5] we rely on the translation lookaside buffers (TLB). We tag TLB entries with extra EID bits, so that processes and enclaves can only use translations that were created specifically for them. An added benefit of tagging the TLB is that the TLB does not need to be flushed on context switches. However, since we rely on the OS to perform memory management, a malicious OS could map the enclave into its own memory space and directly read information stored in the caches. To prevent this, we validate the TLB entries using the DMG sanitizer. This module is identical to the access controller, with one important distinction: instead of checking whether addresses match any of the rules, the DMG sanitizer checks the values of the page table entries (PTE). With this modification, if an OS attempts to create a PTE with an address that points to enclave memory, the DMG sanitizer will trigger an interrupt. The final modification is tagging cache lines with a single Enclave/Non-enclave bit. This modification prevents the OS from bypassing the TLB and directly reading cached data by operating on the physical address space. The caches are modified so that when the TLB is off, cache hits can occur only on cache lines tagged as containing non-enclave data.

VI. Related Work

Enclave security architectures. We group existing enclave architectures depending on whether some of the underlying security mechanisms must be implemented in the processor logic or not. In the former group, Intel SGX [5] provides user-space enclaves managed by the OS based on security mechanisms mainly implemented through processor microcode and a set of hardware changes at the page table walker which perform enclave access control during

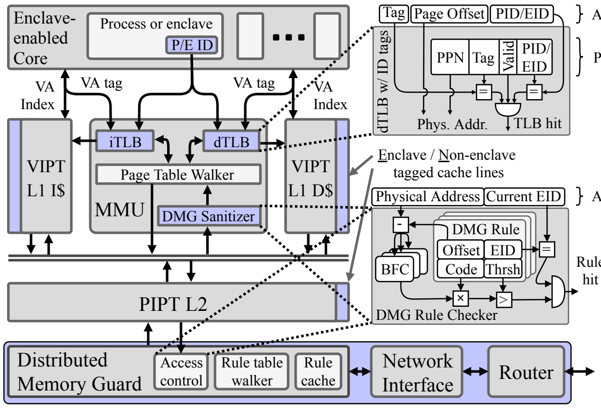


Fig. 6: Modifications at the DMG and security-enhanced cores.

the virtual-to-physical address translation. Academic extensions of SGX (e.g., [22], [23]) also do not change the basic principle of how access control is achieved in SGX. Sanctum [6], which shares many design goals with SGX, also performs access control on the virtual address translation, implemented with additional circuitry around the page table walker. AMD SEV [4] extends the memory controller by an encryption engine to transparently encrypt all data stored in the DRAM, whereas each enclave (comprising of a virtual machine) gets an Address Space ID (ASID) linked to a unique encryption key. Inside the processor, access control is performed by (unmodifiable) microcode to prevent a malicious assignment of ASIDs to enclaves. ARM TrustZone introduces a new processor mode (secure mode) to separate the enclave from the non-enclave data in the processor registers and caches. Proposals based on TrustZone (e.g., [7], [24]) require the same on-core hardware extensions. Since all presented approaches implement enclave security mechanisms in the processor logic, unmodifiable third-party processors cannot be supported in a NoC scenario since they would pose a security threat to the system.

In the latter group, Keystone [8] uses the PMP to perform access control on physical memory addresses outside of the processor logic without requiring additional on-core modifications. CURE [9], which targets SoC-based platforms, moves access control to the system bus which, in contrast to Keystone, enables a secure unencrypted communication between enclaves and devices. Both approaches use a simple rule design for access control which demands the allocation of contiguous memory chunks which, as we show in Section IV, requires to move many memory pages during memory allocation.

Network-on-Chip security. NoCs security has been an active field of research, however, many of the works focus on NoC side-channel attacks and defenses, e.g., [20], [17], and thus have a different focus than our work. When designing security architectures for NoC platforms, some authors propose to deploy encryption engines to the NoC to encrypt the communication between all computing nodes [25]. This is orthogonal to our work since encryption is only justifiable when physical hardware attacks on the NoC bus are considered. The NoC security research most related performs access control directly at the network interface, e.g., [26]. However, all these approaches have a coarse-grained view on the NoC platform and define complete computing nodes as either secure or non-secure entities. Moreover, IOMMU-like hardware components are integrated into the network interfaces which induces a large hardware overhead on large-scale NoC platforms. In this work, we design a more lightweight access control component (DMG) which allows, for the first time, to enable scalable enclave computing NoC platforms.

VII. Conclusion

In this work, we tackle the problem of enabling secure execution of enclaves on NoC-based architectures. We show that physical memory fragmentation makes executing enclaves in the same memory zone as the rest of the system infeasible. We propose a solution that allows enclaves to protect fragmented physical memory through flexible non-contiguous rules. We introduce a new hardware primitive for designing isolation-based secure architectures. This primitive, called the *Distributed Memory Guard (DMG)*, enables rule-based, hardware-rooted, fragmentation-aware memory region access controls.

Acknowledgement

We thank our anonymous reviewers for their valuable and constructive feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG) SFB 1119 CROSSING/236615297. This research is partially supported by Brookhaven National Laboratory under the US Department of Energy (DOE) Office of Science, Office of Basic Energy Sciences, contract DE-SC0012704.

References

[1] Sandvine Incorporated, "2019 Global Internet Phenomena Report," 2020.
[2] Woodruff et al., "The CHERI capability model: Revisiting RISC in an age of risk," in *ISCA*, 2014.
[3] Song et al., "HDFI: Hardware-assisted data-flow isolation," in *S&P*, 2016.
[4] Kaplan et al., "AMD memory encryption," 2016.
[5] McKeen et al., "Innovative Instructions and Software Model for Isolated Execution," in *HASP*, 2013.
[6] Costan et al., "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," in *USENIX Security*, 2016.
[7] Brasser et al., "SANCTUARY: ARMing TrustZone with User-space Enclaves," in *NDSS*, 2019.
[8] Lee et al., "Keystone: An open framework for architecting trusted execution environments," in *EUROSYS*, 2020.
[9] Bahmani et al., "CURE: A Security Architecture with CUstomizable and Resilient Enclaves," 2020.
[10] F. et al., "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware."
[11] Mauerer, *Professional Linux Kernel Architecture*. GBR: Wrox Press Ltd., 2008.
[12] Craciunas et al., "A Compacting Real-Time Memory Management System," in *USENIX ATC*, 2008.
[13] Werner et al., "Transparent memory encryption and authentication," in *FPL*, 2017.
[14] Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *CRYPTO*, 1996.
[15] Biehl et al., "Differential fault attacks on elliptic curve cryptosystem," in *CRYPTO*, 2000.
[16] Tang et al., "CLKSCREW: exposing the perils of security-oblivious energy management," in *USENIX Security*, 2017.
[17] Reinbrecht et al., "Side channel attack on NoC-based MPSoCs are practical: NoC Prime+ Probe attack," in *SBCCI*, 2016.
[18] Gras et al., "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *USENIX Security*, 2018.
[19] Yarom et al., "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security*, 2014.
[20] Wassel et al., "Networks on chip with provable security properties," *Micro*, 2014.
[21] Wang et al., "Efficient timing channel protection for on-chip networks," in *NOCS*, 2012.
[22] Shih et al., "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *NDSS*, 2017.
[23] Ahmad et al., "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX," in *NDSS*, 2018.
[24] Z. et al., "Sectee: A software-based approach to secure enclave architecture using tee," in *CCS*, 2019.
[25] Kinsy et al., "Hermes: Secure heterogeneous multicore architecture design," in *HOST*, 2017.
[26] Porquet et al., "NoC-MPU: A secure architecture for flexible co-hosting on shared memory MPSoCs," in *DAC*, 2011.

E

CHUNKED-CACHE: On-Demand and Scalable Cache Isolation for Security Architectures (NDSS'22)

[62] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **CHUNKED-CACHE: On-Demand and Scalable Cache Isolation for Security Architectures**. In *Symposium on Network and Distributed System Security (NDSS)*, 2022. CORE Rank A*. Appendix E.

CHUNKED-CACHE: On-Demand and Scalable Cache Isolation for Security Architectures

Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, Emmanuel Stapf
Technical University of Darmstadt, Germany
{ghada.dessouky, pouya.mahmoody, ahmad.sadeghi, emmanuel.stapf}@trust.tu-darmstadt.de

***Abstract*— Shared cache resources in multi-core processors are vulnerable to cache side-channel attacks. Recently proposed defenses such as randomized mapping of addresses to cache lines or well-known cache partitioning have their own caveats: Randomization-based defenses have been shown vulnerable to newer attack algorithms besides relying on weak cryptographic primitives. They do not fundamentally address the root cause for cache side-channel attacks, namely, mutually distrusting codes sharing cache resources. Cache partitioning defenses provide the strict resource partitioning required to effectively block all side-channel threats. However, they usually rely on way-based partitioning which is not fine-grained and cannot scale to support a larger number of protection domains, e.g., in trusted execution environment (TEE) security architectures, besides degrading performance and often resulting in cache underutilization.**

To overcome the shortcomings of both approaches, we present a novel and flexible set-associative cache partitioning design for TEE architectures, called CHUNKED-CACHE. The core idea of CHUNKED-CACHE is to enable an execution context to "carve" out an exclusive *configurable chunk* of the cache if the execution requires side-channel resilience. If side-channel resilience is not required, mainstream cache resources can be freely utilized. Hence, our proposed cache design addresses the security-performance trade-off practically by enabling efficient selective and on-demand utilization of side-channel-resilient caches, while providing well-grounded future-proof security guarantees. We show that CHUNKED-CACHE provides side-channel-resilient cache utilization for sensitive code execution, with small hardware overhead, while incurring no performance overhead on the OS. We also show that it outperforms conventional way-based cache partitioning by 43%, while scaling significantly better to support a larger number of protection domains.

I. INTRODUCTION

The outbreak of micro-architectural attacks has demonstrated the crucial implications of performance-boosting processor optimizations on the security of our computing platforms [54], [1], [60], [56], [52], [65], [100], [31], [28], [27], [58], [4], [3], [88], [68], [90], [92], [16], [17], [81], [15]. One of the most popular features, and also the subject of many recent attacks, are shared resources such as caches. Caches provide orders-of-magnitude faster memory accesses and large last-level-caches (LLCs) are usually shared across multiple processor cores to maximize utilization.

The Problem with Caches. When a sensitive (victim) and malicious (adversary) application run simultaneously on different cores and share the LLC, cache side channels can be exploited by the adversary to leak sensitive information, such as private keys. The timing difference between a cache hit and miss – which is why caches are used in the first place – is the most commonly exploited side channel to infer the memory access patterns of a victim application [38], [99], [35], [44], [34], [43], [46], [64], [32], [71], [36], [37], [98], [91]. In typical side-channel attacks [71], [43], [46], [64], [38], [99] the adversary deduces the victim's memory access patterns by exploiting that both the victim and adversary compete for shared set-associative cache resources, which are designed in such a way that a larger number of memory lines are mapped to a smaller number of cache ways/entries in each cache set.

Besides compromising cryptographic implementations [7], [64], [71], [99], more recent attacks have had even stealthier impact such as bypassing address space layout randomization (ASLR) or leaking privacy-sensitive human genome indexing computation [34], [32], [14], [35], [36], leaving millions of platforms vulnerable. Even trusted execution environment (TEE) security architectures which aim to protect sensitive services by compartmentalizing them in isolated execution contexts, called *enclaves*, e.g., Intel SGX [41], [21] or ARM TrustZone [5], have been shown vulnerable to these attacks, thereby undermining their acclaimed privacy and isolation guarantees [14], [83], [69], [30], [59], [101]. This is alarming since TEE architectures are now widely deployed by major cloud providers, e.g., Microsoft Azure, Google Cloud, Alibaba Cloud and IBM Cloud, to offer *confidential computing*, where sensitive workloads are protected in enclaves.

The Problem with Recent Cache Defenses. To mitigate cache side-channel attacks, various approaches have been proposed over the years. These solutions range from time-constant cryptographic implementations [26], [25], [55] to software- and hardware-based approaches that modify the cache organization itself. The latter can be broadly classified into either cache partitioning [29], [94], [51], [61], [23], [51] or randomization-based techniques [63], [89], [77], [78], [96], [87] that attempt to obfuscate the relationship between the memory address and the cache location to which it is mapped.

More recently, various schemes for a randomized memory-to-LLC mapping, such as CEASER, ScatterCache, and Phantom-Cache [89], [77], [78], [96], [87] have been proposed to mitigate these attacks by obfuscating the adversary's view of which cache lines actually get evicted. However, such defenses continue to evict cache lines from a small number of locations in a shared cache, thus cache set-based conflicts essentially still occur. While these defenses were shown effective against the eviction set construction algorithms and techniques at the time, subsequent more efficient eviction set construction algorithms [78] were able to undermine them. Consequently,

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23110
www.ndss-symposium.org

enhancements to these defenses were proposed [78], only to be rendered ineffective again by yet another attack vector, e.g., weak low-latency cryptographic primitives [74], [10], or alternative attack techniques that exploited design/implementation flaws in the proposed defenses [86].

Caught in an arms race, randomization-based defenses remain as good as the best known attack technique at the time and are constructed to mitigate very specific side channels and attack strategies [12], with no future-proof and well-grounded security guarantees. They only make the attacks computationally more difficult, but do not address their fundamental root cause, i.e., sharing set-associative caches across mutually distrusting processes. These schemes also assume that all execution contexts require side-channel resilience without providing mechanisms for a selective configuration of side-channel-resilience, thus, taxing the entire system with the resulting performance impact. In practice, however, only a small portion of the workload is usually security-/privacy-sensitive and requires this sophisticated security guarantee.

On the other hand, strict partitioning approaches promise well-grounded security guarantees due to their cache isolation across different execution contexts. However, these approaches usually rely on conventional way-based partitioning [6], [57], [94], [51], [23], and thus, are not fine-grained, cannot scale with an increasing number of execution contexts and large LLCs, or do not provide support for shared memory.

With these limitations in mind, we argue that a more future-proof and practical approach for side-channel resilient cache computing is to address the root cause of these attacks, namely, sharing set-associative cache structures across mutually distrusting execution contexts. Meanwhile, performance, usability, flexibility and scalability should still be preserved. We further observe that, in practice, cache side-channel resilience is most prominently a concern in dedicated security architectures, e.g., TEE security architectures. Thus, it is crucial to develop side-channel-resilient cache designs that cater for the security/functionality requirements of these architectures, e.g., with integrated support for enabling the side-channel resilience (and the performance cost) only for specific execution contexts that require it.

Our Goals. In this work, we aim to selectively enforce clean partitioning of the cache resources across mutually distrusting execution contexts that require side-channel resilience, such that all side channels are blocked (including stealthy cache occupancy channels [84] which are not mitigated by recent works [96], [23]), while maintaining the desired performance requirements.

To address this performance-security trade-off, we propose a new cache design for TEE security architectures, which we call CHUNKED-CACHE, that enables each execution context or domain to "carve" out its exclusive cache *sets*, if desired. These sets essentially constitute an independent set-associative cache, which we call the domain's *cache chunk*, that this domain can utilize exclusively but fully and efficiently, unlike in cache partitioning, e.g., way-based partitioning. A domain can flexibly request and configure 1.) whether it requires side-channel-resilient cache utilization, 2.) for which memory regions, and 3.) the required capacity of this exclusive side-channel-resilient *cache chunk*. Memory accesses by a domain that requires side-channel-resilient cache utilization are mapped exclusively to its *cache chunk*, while mainstream cache resources are freely and conventionally utilized whenever side-channel-resilience is not required. Enabling this on-demand flexibility per domain *practically* requires addressing multiple key challenges. Firstly, efficient design mechanisms are required to configure the memory-to-set mapping at runtime for each domain depending on its chunk capacity, while preserving conventional cache behavior for the rest of the execution. Secondly, it must be ensured that the operating system performance is not degraded as cache sets get allocated exclusively to domains. Finally, seamless support must be provided for shared memory between domains to meet the security and functionality requirements of different sensitive applications.

Our Contributions. Our main contributions are as follows:

- We present CHUNKED-CACHE, a novel cache architecture for TEE security architectures, which enables a selective, flexible and scalable configuration of side-channel resilient caches for execution domains, without degrading the OS performance.
- We address the performance-security trade-off by enforcing clean cache partitioning that blocks all cache side channels by allocating exclusive cache chunks for different domains. In doing so, future-proof and solid security assurances are guaranteed, while still preserving performance, functionality and compatibility requirements.
- We extensively evaluate the cycle-accurate performance overhead of CHUNKED-CACHE for compute-intensive SPEC CPU2017 workloads and I/O-intensive real-world applications. We show that it outperforms shared cache utilization in some cases, that the OS performance even improves owing to CHUNKED-CACHE's flexible cache utilization, and that CHUNKED-CACHE outperforms partitioning (way-based) by 43% while also scaling better to support a larger number of protection domains.
- We implement and evaluate a hardware prototype of CHUNKED-CACHE. We show that it incurs a minimal 2.3% memory overhead relative to a 16 MB LLC, 1.6% logic overhead relative to a single-core RISC-V processor, and 12.3% LLC power consumption overhead.

II. CACHE ATTACKS & DEFENSES

Next, we briefly introduce recent cache side-channel attacks that are relevant for our work and a summary of the shortcomings of recent defenses that our work overcomes.

A. Cache Side-Channel Attacks

Cache side-channel attacks have been shown to constitute a profound threat that underlies popular attacks such as Spectre [54] and Meltdown [60], besides threatening a wide spectrum of platforms and architectures [59], [64], [43], [102], and even TEE architectures [14], [83], [69], [30], [59], [101]. The attacks usually work by provoking controlled evictions of the victim's cache line, such that the inherent information leakage from the access-timing difference between cache hits and misses can be exploited by the adversary. This can be achieved using three main approaches:

- Access-based approaches where the target address is explicitly accessed and flushed [38], [99], [35], [44], [34].

- Conflict-based approaches where the adversary triggers a controlled cache contention in the same cache set of the target address to evict the corresponding victim cache lines [71], [43], [46], [99], [64], [98], [24], [32], [71], [37], [91], [7], [11].
- Occupancy-based approaches [84] where the adversary observes an eviction of its own cache lines and uses this information to infer the size of the victim's working set.

B. *Recent Defenses and their Shortcomings*

Various defenses against side-channel attacks have been proposed, focusing on access-based and conflict-based attacks.

Side-channel Resilient Implementation. This aims at implementing algorithms, e.g. cryptographic algorithms, in a time-constant (thus side-channel-resilient) fashion [42], [8]. Time-constant algorithms vary between hardware platforms [19] and require considerable effort that is not generalizable and scalable for all software.

Attack Detection. Other approaches aim to detect attacks in progress by observing hardware performance counters (e.g., on cache miss rates) [18], [72] and killing the suspicious process. However, being based on heuristics, attacks can only be discovered with a certain probability and no guaranteed protection is provided. Moreover, some attacks have been shown to not cause an abnormal cache behavior [35].

Noisy Measurements. Another group of defenses aims to impede a successful attack by preventing the adversary from performing precise time measurements, e.g., by restricting the access to timers [71], [73], [66], by injecting noise into the system [93], [40] or deliberately slowing down the system clock [39], [67]. However, workarounds have been found to create timers [82] or to perform attacks without relying on timers [24]. Moreover, such defenses cannot protect TEE architectures since they assume a strong adversary that can compromise the OS kernel and circumvent such restrictions.

Cache-level Defenses. Other approaches tackle the side-channel problem directly where it originates, i.e., at the cache level. These defenses fall under one of two paradigms: 1.) randomized cache line mapping to make the attacks computationally impractical [89], [77], [78], [96], [87], [95], [63], [62] or 2.) cache partitioning to provide strict isolation [29], [50], [101], [61], [22], [33], [103], [47], [97], [57], [6], [94], [51], [95], [23]. We discuss the works most related to CHUNKED-CACHE in more detail in Section VII.

Randomization-based defenses cannot provide comprehensive future-proof security guarantees, e.g., advances in attack strategies and minimal eviction set construction techniques, besides alternative attack techniques have been shown to undermine such defenses [78], [12], [75], [74], [86]. Moreover, many rely on cryptographic primitives which have been shown vulnerable to cryptanalysis, while deploying more secure primitives would further degrade performance [10], [74].

Cache partitioning defenses provide strict resource isolation which allows to give solid security guarantees on side-channel protection. However, existing partitioning defenses suffer from high performance penalties, restrictive and inflexible cache utilization [95] and their inability to scale with a larger number of protection domains [94], [51], [33]. Several approaches do not directly cater for the use of shared libraries [29], [94], are architecture-specific [47], [97] or do not defend against occupancy-based attacks. Memory page coloring approaches [22], [50], [29] are impractical since they require invasive modifications of the memory management of commodity software and cannot sufficiently support Direct Memory Access (DMA). Most importantly, existing partitioning defenses to date apply their side-channel cache protection for the entire execution workload, impacting overall system performance, which is not even required in most scenarios.

To fundamentally address all these shortcomings, we propose a modified cache microarchitecture, which we call CHUNKED-CACHE, that provides strict, yet configurable partitioning across the mutually distrusting execution domains. For each domain, CHUNKED-CACHE carves out and isolates an exclusive cache share only as the domain requires. This effectively mitigates all interference across domains, thus, defending against even stealthy cache occupancy attacks unlike recent cache defenses, while activating side-channel resilience only for sensitive execution domains that require it. All other execution domains can freely utilize mainstream cache resources at the same performance or even improved performance than conventional non-secure cache sharing.

III. SYSTEM & ADVERSARY MODEL

In the following section, we describe our assumptions regarding the system and adversary model.

A. *System Model*

CHUNKED-CACHE targets computing systems which implement a TEE security architecture and contain a set-associative cache architecture. In the following, we first present our standard assumptions regarding the cache architecture, followed by our assumptions on the TEE security architecture which are aligned with existing academic [22], [57], [13], [6] and industry solutions [41], [45], [5].

Cache Architecture. In CHUNKED-CACHE, we assume a typical modern set-associative cache architecture with multiple cache levels, where some cache levels are core exclusive (typically L1 and L2) and others shared between multiple cores (L3), whereby the L3 can be a sliced cache, e.g., sliced Intel LLCs. While CHUNKED-CACHE can be deployed to provide partitioning for smaller L1 and L2 caches in principle, we assume, however, that core-exclusive caches are flushed at context switching (similar to most recent TEE architectures [6], [22], [57]), and thus, that CHUNKED-CACHE is deployed for the last-level L3 cache. Moreover, we assume that the cache controller can be configured via dedicated configuration registers, in line with typical platforms.

TEE Architecture. We assume that the computing systems which deploy CHUNKED-CACHE implement a TEE architecture. TEE architectures already have established mechanisms for protecting sensitive code in compartmentalized execution contexts called *enclaves* or **I**solated **D**omains (I-Domain), as we refer to them in this work. All non-sensitive code which does not require enhanced protection is consolidated in a **N**on-**I**solated **D**omain (NI-Domain). The domains are also each assigned a unique identifier (domain ID). The separation between the I-Domains and the NI-Domain is enforced by access control mechanisms already implemented in the TEE architectures, e.g., at the MMU in Intel SGX [41] or Sanctum [41], at the system bus in CURE [6] or by the Physical Memory Protection (PMP) unit in Keystone [57]. The access control mechanisms are either configured by microcode [41], [45] or by a small software component which consists only of a few thousand lines of code (to be formally verifiable) and which runs in the highest software privilege level of the system [22], [57], [13], [6], [5]. We refer to this component as a *trusted software component*. The trusted software component is also responsible for all other security-sensitive operations, e.g., assigning the domain IDs, and, in the case of CHUNKED-CACHE, configuring our novel protection mechanisms in the cache controller which we describe in detail in Section IV.

Although I-Domains are security-sensitive, they might still require to share data with another domain, e.g., to enable communication with the operating system. Thus, TEE architectures typically provide the possibility to mark parts of an I-Domain's memory as *shared*, whereby this information is again managed by the trusted software component. In many TEE architectures, e.g., TrustZone [5], CURE [6] or AMD SEV [45], security-relevant metadata, which is required to perform access control, is sent as part of every memory request. For CHUNKED-CACHE we assume the same, namely, that the domain ID of the domain issuing a memory access request and the information whether the requested memory address is *shared* or *non-shared*, are sent within the memory request.

B. Adversary Model

Since we focus on the deployment of CHUNKED-CACHE on systems with TEE architectures, we assume the same strong adversary model where the operating system kernel and hypervisor are untrusted [22], [57], [13], [6], [41], [45], [5].

With regard to cache side-channel attacks, we assume the adversary has access to the CHUNKED-CACHE specification and is able to mount access-based and conflict-based side-channel attacks, which are the most sophisticated and applicable cache attacks (cf. Section II-A), to leak information about a sensitive execution domain (I-Domain). Since the adversary is also able to control the OS kernel, we assume a worst-case scenario where an adversary can easily mount the described attacks, i.e., has knowledge about the CHUNKED-CACHE design and specs, and knows the virtual to physical address mapping of the victim domain. Moreover, the adversary can mount attacks from all privilege levels (except the highest privilege level that contains the trusted software component), has access to precise timing measurements and eviction instructions (e.g., `clflush`), can attack from the same CPU core executing the victim domain or a different core (cross-core), freely interrupt the victim domain and even keep the system noise to a minimum. In contrast to related work [89], [77], [78], [96], [87], we also consider the stealthier cache occupancy-based attacks (cf. Section II-A). Collision-based attacks [11], which exploit cache collisions at the victim caused by the victim's own cache utilization, are, aligned with related work, kept out of scope. Collision-based attacks have not been widely shown and are very specific to particular software implementations (e.g., table-based).

Apart from cache side-channel attacks, an adversary who compromises the OS kernel has full control over the memory management and thus, can easily map physical memory pages of a victim domain into its own memory. This allows an adversary to perform rogue cache accesses to sensitive data *directly* without the need of a cache side channel.

In line with related work [29], [94], [51], [61], [23], [95], [63], [89], [77], [78], [96], [87], we do not consider physical attacks on caches, e.g., physical side-channel attacks [53], fault injection attacks [9], and attacks that exploit hardware flaws [88], [48], [76]. We do not consider denial-of-service attacks from a security point of view. However, to avoid the performance impact on the OS, CHUNKED-CACHE ensures that a certain amount of cache resources are always available to the OS (described in Section IV). Based on our system model (Section III-A) , we assume that the adversary cannot compromise the trusted software component.

IV. CHUNKED-CACHE DESIGN

We first describe the high-level idea of CHUNKED-CACHE, a novel cache microarchitecture that provides flexible and on-demand assignment of cache resources to execution domains (Section IV-A). We follow with a detailed explanation of our design (Section IV-B) and the required cache tag store and cache controller modifications (Section IV-C).

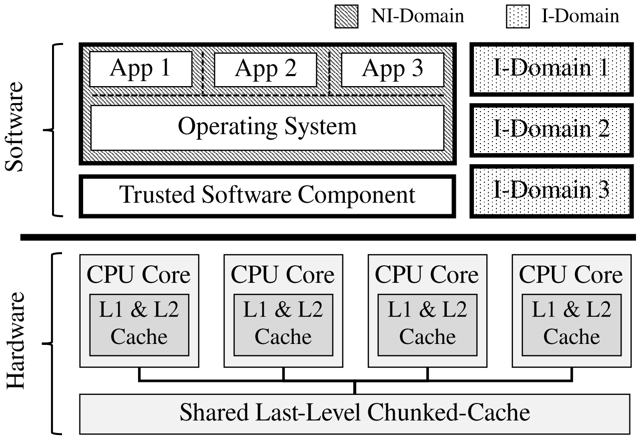


Fig. 1. Computing system with TEE architecture and CHUNKED-CACHE as the shared last-level cache.

A. High-Level Design

In Figure 1, we show how CHUNKED-CACHE is integrated as the last-level cache in a computing system which implements a TEE architecture, aligned with our system model detailed in Section III-A. Figure 2 steers the focus to the design of CHUNKED-CACHE itself and illustrates its architecture abstractly. As described in Section III-A, all TEE architectures provide built-in mechanisms to protect sensitive code in **I**solated **D**omains (I-Domains), whereas non-sensitive code is running in a **N**on-**I**solated **D**omain (NI-Domain).

Each active domain (NI-Domain and I-Domains) is uniquely identified by an ID: `DID`. The operating system (OS) and all workloads which do not require protection (and are combined in the NI-Domain) are assigned the `DID` 0 by default. Every I-Domain can request exclusive cache resources

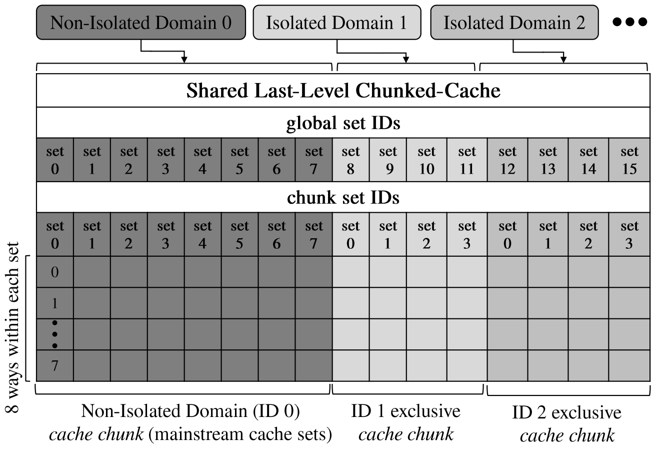


Fig. 2. Chunked-Cache high-level design: each domain gets an exclusive *cache chunk* allocated on-demand.

of desirable capacity, forming the domain's exclusive *cache chunk*, that is only utilized by the owner domain. The NI-Domain utilizes the cache sets which are not exclusively allocated to I-Domains, which we call *mainstream* cache sets.

Each I-Domain requests its dedicated *cache chunk* consisting of the required number of cache sets, e.g., I-Domain 1 in Figure 2 requested 4 sets. Thus, at I-Domain 1 setup, 4 available (unallocated) sets are located in the cache (sets with global IDs 8-11 here) and allocated to I-Domain 1 such that they form its *cache chunk*. The allocated sets are mapped to I-Domain 1's chunk set IDs 0-3, and they are used to exclusively cache all and only memory accesses issued by I-Domain 1. Enabling each I-Domain to request its desired *cache chunk* capacity exclusively provides strict partitioning and completely isolates its cache utilization on-demand. Besides enabling selective cache-based side-channel resilience, this also allows that each I-Domain acquires the performance that corresponds to the cache capacity it has requested, without any competition from other workload. In contrast to partitioning schemes [61], [57], [6], [94], [51] that provide each domain with only 1 or 2 ways within each set of the full cache structure, Chunked-Cache also partitions the cache but more efficiently. Chunked-Cache carves out a full *cache chunk* (with all its ways per set) of configurable capacity for the I-Domain and configures all its memory accesses to be mapped to the *cache chunk*, thus promising maximum and unshared utilization of the allocated *cache chunk*. We show in Section VI that Chunked-Cache provides better performance and enhanced scalability than partitioning schemes.

By allowing each I-Domain a custom and configurable *cache chunk* capacity on-demand, in contrast to fixed allocation, Chunked-Cache enables an adaptive security-performance trade-off in the cache microarchitecture. On one hand, non-sensitive workload can be allowed to freely utilize the shared mainstream cache resources. On the other hand, if side-channel resilience is a concern, a *cache chunk* with default capacity can be allocated to each I-Domain without any further intervention from the developer. Only if the developer requires to further optimize the performance of the workload in a particular I-Domain, then the *cache chunk* capacity (its number of sets) can be accordingly calibrated, i.e., assigning an I-Domain more cache resources if affordable/available.

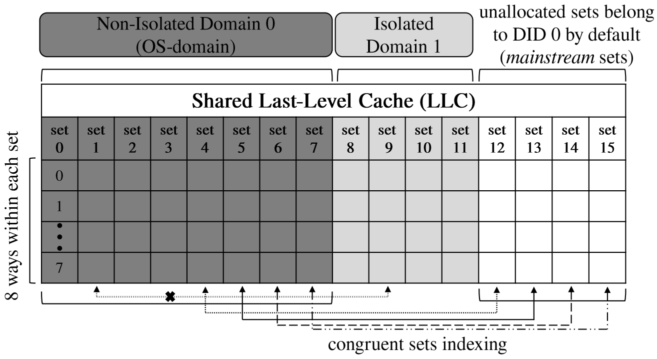
B. Design Details of Chunked-Cache

In the following, we discuss the key design goals and challenges of Chunked-Cache, and the mechanisms we propose to achieve them.

Configurable Per-Domain Isolation Modes. One of our key design goals for Chunked-Cache is to support configurable cache isolation modes that provide different security guarantees, thus catering for different use cases and their requirements. In line with the design paradigm of TEEs, it is not reasonable to assume that all workloads require cache isolation and side-channel resilience. Thus, in Chunked-Cache, we provide 2 different Isolation Modes that each I-Domain can selectively configure for the workload it protects: 1.) Mainstream-Cache Mode: where cache isolation and side-channel resilience is not a security requirement, and thus, the I-Domain can utilize the mainstream cache. However, the cached I-Domain data must still be protected from malicious OS accesses. 2.) Exclusive-Cache Mode: where cache isolation is required since side-channel resilience is a security requirement and thus, an exclusive *cache chunk* is required by this I-Domain. The latter mode is configured for I-Domain 1 and I-Domain 2 shown in Figure 2. In addition to the Isolation Mode, the I-Domain can also configure its Shared Memory settings, i.e., if it requires to share memory regions (and thus cache lines) with the OS, e.g., when using OS services. To cache shared memory, the mainstream cache that the OS uses is utilized. Typically, the developer of the workload decides which Isolation Mode an I-Domain uses and identifies which memory regions need to be shared, which is on par with the requirement in TEE architectures where the developer must identify the security-sensitive parts of the overall workload [41], [5], [22]. If a developer is not sure whether cache side-channel attacks are a threat, the Exclusive-Cache Mode should be selected out of caution. At setup, an I-Domain configures: 1.) the desired Isolation Mode for its cache utilization and 2.) its Shared Memory regions if required. This metadata is securely configured by the trusted component (as shown in Figure 1). The Isolation Mode is communicated to the cache controller at domain setup, whereas the Shared Memory information is transmitted at every memory request, aligned with our assumed system model (Section III-A).

Mainstream Cache vs. Shared Memory Support. When an I-Domain is in Mainstream-Cache Mode, it uses the mainstream cache sets also used by the OS (`DID` 0). To prevent a malicious OS from mapping the memory of an I-Domain in its own memory space and accessing it directly in the cache, Chunked-Cache requires that cache lines are tagged with the domain ID `DID`. The hardware mechanisms integrated into the Chunked-Cache controller enforce this tagging when caching the data, and that only the owner domain which cached the data can access it. Being hardware managed, the OS has no means to modify the `DID` stored in the cache lines.

When an I-Domain is also sharing memory with the OS, the corresponding cache lines for the defined Shared Memory



Each memory access by DID 0 (OS Domain) is mapped to a set in the OS *principal cache chunk* as well as congruent sets if unallocated (*mainstream* cache sets)

Fig. 3. CHUNKED-CACHE OS-specific chunk set indexing.

regions are cached in the mainstream cache sets, and are to be accessed by both the owner domain and the OS. To support that, cache lines need to be tagged with an additional SHARED flag that indicates whether the cache line is shared with the OS. For typical TEE architectures, the developer of the workload protected in the I-Domain configures which of its memory regions are to be shared.

EXCLUSIVE-CACHE MODE Chunk Set Indexing. The *index* bits of a memory address are used to locate the cache set to which it is mapped. In a conventional cache, the number of index bits is fixed and depends on the number of sets the cache supports. However, for CHUNKED-CACHE to support *cache chunks* of different sizes for different domains, *configurable set indexing* is required.

When an I-Domain is in EXCLUSIVE-CACHE MODE and requests a number of cache sets for its *cache chunk*, the number of set index bits that will be used to map its memory lines has to be computed individually for this domain. Therefore, the cache controller keeps track of the global IDs of sets which constitute the *cache chunk* (Figure 2), and the index bits for each domain. When a memory access is issued by a domain, this metadata is looked up, and the pertinent *cache chunk* sets correctly indexed. Moreover, when an I-Domain is torn down and its sets are de-allocated, the relevant metadata needs to be updated accordingly, besides flushing and invalidating the cache lines. CHUNKED-CACHE also enables support for dynamic cache allocation, i.e., allocating additional cache sets to an I-Domain's *cache chunk* at runtime and reconfiguring the index bits accordingly. In Section IV-C, we describe how the cache microarchitecture and controller are modified to enable this configurability efficiently.

NI-Domain Chunk Set Indexing. Another design challenge in CHUNKED-CACHE is managing the sets allocated to the OS, which represents the NI-Domain with DID 0, such that both flexibility as well as maximum utilization (as in an unmodified insecure cache architecture) are preserved. At bootup, when no domains are set up yet besides the OS, the OS should ideally be able to utilize all the available cache capacity, i.e., all cache sets are allocated to the OS by default. We refer to these as the *mainstream* cache sets. Then, once domains are set up and request exclusive cache sets, these get "torn away" from the OS's cache and are allocated to the domains. This would, however, incur an impractical performance degradation for the OS since every time some of the OS's cache resources are allocated to another domain, its own capacity is changed, and so would its set indexing. This renders all memory lines already cached by the OS inaccessible unless complicated remapping is performed. Essentially, the OS would need to cache these memory addresses once again, thus suffering a high number of cold misses every time a new domain is set up and subjecting the OS to an unreasonably high performance overhead.

To avoid this performance penalty on the OS, the OS is allocated a fixed (sufficiently large) number of the cache sets in CHUNKED-CACHE which remain always dedicated to the OS, while still allowing it to utilize the other cache sets so long as they remain unallocated. We demonstrate this in Figure 3 where the OS is always allocated a fixed number of 8 sets (0-7) which form its principal *cache chunk*. Since the 8 sets are always available for the OS, the memory address indexing and the number of index bits do not change at runtime. In other words, no OS memory lines cached in this principal *cache chunk* must ever be flushed out when any other domain requests to allocate additional cache sets, since the OS *cache chunk* sets are never torn away from the OS. However, the OS can still utilize unallocated sets (sets 12-15) in parallel until they get allocated to another domain, thus also guaranteeing maximum utilization of the available cache resources. This works by indexing cache sets in parallel which are congruent to the set to which a memory address is mapped. In Figure 3, 3 index bits are required to map a memory address to the correct set for a *cache chunk* of size 8 sets. Thus, if the index bits, e.g., map to set 4, then set 12 can also be utilized by the OS (set ID + OS *cache chunk* size) to cache that memory line. The same applies for memory lines that are mapped to sets 5, 6 and 7; they also map to sets 13, 14 and 15, respectively. However, memory lines mapped to sets 0-3 cannot utilize the congruent sets 8-11 because these are already allocated to I-Domain 1.

C. Cache Tag Store & Cache Controller

Cache lines need to be additionally tagged with the domain ID (DID) bits as well as a 1-bit SHARED flag bit to enforce access control and moderate sharing with the NI-Domain. For instance, to support 16 parallel active domains, we require to extend the cache tag store with 4 bits to represent the DID. We emphasize that the CHUNKED-CACHE design does not limit the number of parallel domains to 16; a larger number is possible but increases the hardware overhead of CHUNKED-CACHE (but only linearly). Moreover, the number of domains only limits how many domains can be simultaneously active on the system. It does not limit how many applications can be protected in I-Domains on the system in general.

To support the configurable set indexing, the allocation/de-allocation of cache sets to different I-Domains and to differentiate between OS (NI-Domain) cache accesses vs. I-Domain accesses, 2 table structures are required by the CHUNKED-CACHE controller which are shown in Figure 5. The CACHE SET STATUS TABLE (CST) is a 1-bit vector that is indexed by the global set ID (SID) and that stores the status of each set, i.e., whether it is allocated to a domain. The CST is used to query the status of a set when searching for free cache sets to allocate to an I-Domain.

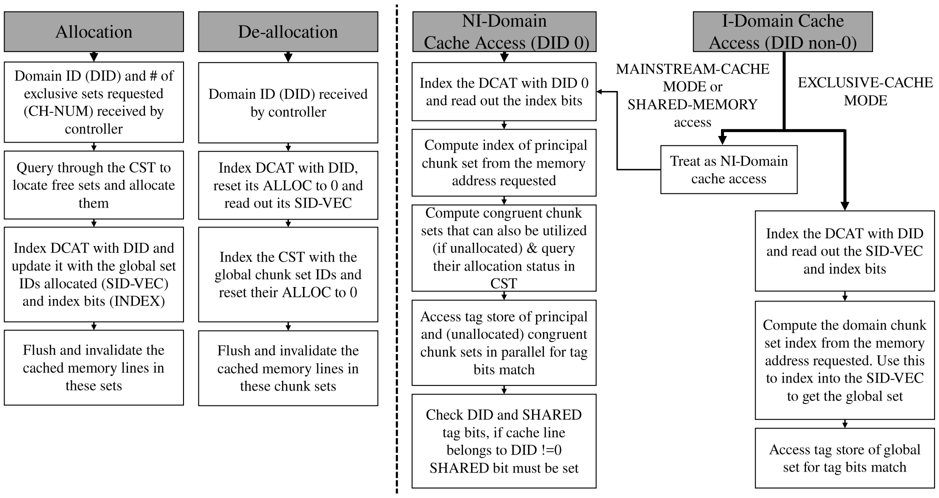
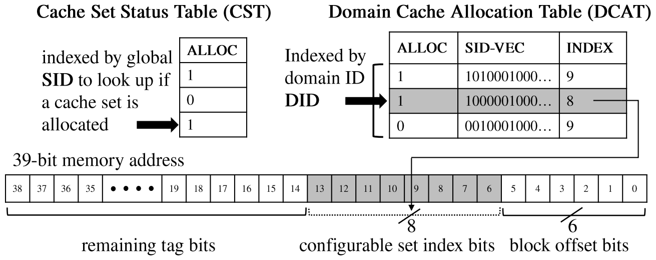

Allocation
Domain ID (DID) and # of exclusive sets requested (CH-NUM) received by controller
Query through the CST to locate free sets and allocate them
Index DCAT with DID and update it with the global set IDs allocated (SID-VEC) and index bits (INDEX)
Flush and invalidate the cached memory lines in these sets
De-allocation
Domain ID (DID) received by controller
Index DCAT with DID, reset its ALLOC to 0 and read out its SID-VEC
Index the CST with the global chunk set IDs and reset their ALLOC to 0
Flush and invalidate the cached memory lines in these chunk sets
NI-Domain Cache Access (DID 0)
Index the DCAT with DID 0 and read out the index bits
Compute index of principal chunk set from the memory address requested
Compute congruent chunk sets that can also be utilized (if unallocated) & query their allocation status in CST
Access tag store of principal and (unallocated) congruent chunk sets in parallel for tag bits match
Check DID and SHARED tag bits, if cache line belongs to DID !=0 SHARED bit must be set
I-Domain Cache Access (DID non-0)
MAINSTREAM-CACHE MODE or SHARED-MEMORY access
EXCLUSIVE-CACHE MODE
Treat as NI-Domain cache access
Index the DCAT with DID and read out the SID-VEC and index bits
Compute the domain chunk set index from the memory address requested. Use this to index into the SID-VEC to get the global set
Access tag store of global set for tag bits match

Fig. 4. CHUNKED-CACHE controller operations for cache chunk allocation, de-allocation and access control.


Cache Set Status Table (CST)
Domain Cache Allocation Table (DCAT)
indexed by global SID to look up if a cache set is allocated
Indexed by domain ID DID
39-bit memory address
38
37
36
35
• • • •
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
8
6
remaining tag bits
configurable set index bits
block offset bits

ALLOC
1
0
1

ALLOC	SID-VEC	INDEX
1	1010001000…	9
1	1000001000…	8
0	0010001000…	9

Fig. 5. CHUNKED-CACHE table structures.

The DOMAIN CACHE ALLOCATION TABLE (DCAT) is indexed by the domain ID DID. It maintains whether this domain is configured by the cache controller (ALLOC), a vector of the global set IDs that form its *cache chunk* (SID-VEC), and the corresponding number of index bits (INDEX) required to map a memory line to the correct set (log_2(number of sets in the *cache chunk*)), as shown in Figure 5.

We describe next how the CHUNKED-CACHE controller performs these cache management operations, i.e., allocation, de-allocation and access control and represent this in Figure 4. The description in Figure 4 only represents the sequence of operations for understanding, but does not reflect the temporal nature of the operations, i.e., whether they occur sequentially or in parallel.

Cache Allocation & De-allocation. When an I-Domain requests to allocate exclusive cache sets, this request (DID, the number of sets (CH-NUM) requested, and the corresponding number of INDEX bits ($\log_2$ CH-NUM) is securely communicated from the trusted component to the cache controller via configuration registers of the cache controller (Section III-A). The DID is looked up in the DCAT to check if it is already allocated and that the maximum sets number allowed per I-Domain is not exceeded. The maximum/minimum limits for I-Domains are configured by the trusted software component, while ensuring that each I-Domain is always assigned at least a minimum *cache chunk* size.

The CST is queried to locate free sets and to allocate them to the I-Domain by flipping the ALLOC bit, until CH-NUM sets are allocated. If CST runs out of free sets, this is communicated back to the trusted component to modify the cache request. Next, the DCAT is indexed with the DID and its metadata updated by updating the INDEX bits and the SID-VEC with the global IDs of the allocated sets.

If a domain requests to de-allocate its cache sets, DCAT is indexed with DID, ALLOC reset and the SID-VEC read out. Next, the CST is indexed with each set ID in SID-VEC and de-allocated. For both allocation and de-allocation, the cached memory lines in the relevant sets are invalidated and flushed (if dirty) to remove potentially malicious data in the allocation case and prevent information leakage in the de-allocation case.

The number of cache sets which are always assigned to the NI-Domain are hardwired, since the circuitry for the parallel tag lookup (described below) must be hardwired.

Cache Access Management. The DID of an incoming cache access request indicates whether it is an access by the NI-Domain (OS domain with DID 0) or an I-Domain. If it is an OS access, then the index bits are fixed, since its number of cache sets are hardwired (no need to look its INDEX bits up in the DCAT). The OS domain is assigned the least significant cache sets by default, thus the SID-VEC is also not needed. The correct set index in the principal chunk is computed from the memory address in the request. Because it is an OS access, congruent cache sets that are not allocated can also be utilized (see Section IV-A). Thus, they are also computed and their ALLOC status queried in the CST to locate the unallocated sets. The tag store of the ways in the principal as well as the congruent sets are looked up in parallel to locate a tag bit match (cache hit), thus, neither impacting performance nor routing delay especially since a large number of principal sets are usually allocated to the NI-Domain which minimizes the number of congruent sets that are looked up in parallel (1 or 2 more sets). The DID and SHARED tag bits are also checked in parallel. If the cache line belongs to a non-zero DID (I-Domain), the SHARED tag bit should be 1 to allow the OS to access it.

For an I-Domain (non-zero DID), if access is requested to a SHARED MEMORY region or if the I-Domain is in MAINSTREAM-CACHE MODE, then the access is treated by the controller as a NI-Domain access where the mainstream and congruent cache sets are accessed. However, at the tag comparison, the issuing DID is checked against the cache line DID to verify that only the owner domain accesses it. If the access is performed in EXCLUSIVE-CACHE MODE, the exclusive *cache chunk* of the domain is accessed. The DCAT is indexed with the DID and the SID-VEC and INDEX bits are read out. The chunk set index is computed and used to index into the SID-VEC to map to the correct global set ID. Then, the tag store is accessed for a tag bits comparison.

CHUNKED-CACHE's design is independent from the implemented cache replacement policy and thus, does not require additional modifications to it. On every cache miss experienced by an I-Domain in EXCLUSIVE-CACHE MODE, a cache line in the corresponding set in the domain's exclusive *cache chunk* is selected for eviction. On cache misses by an I-Domain in MAINSTREAM-CACHE MODE or when accessing SHARED MEMORY, and for all misses by the NI-Domain, a cache line in the corresponding set from the mainstream cache is selected.

V. SECURITY CONSIDERATIONS

In this section, we discuss how CHUNKED-CACHE protects from the adversary described in Section III-B. One key aspect of CHUNKED-CACHE is that its protection capabilities rely on a strict partitioning of cache resources. Thus, in contrast to related work, which rely on probabilistic defenses (e.g., randomized cache line mappings [89], [77], [78], [96], [87]), CHUNKED-CACHE provides certainty that the attacker cannot infer the cache accesses of a victim, if the partitioning is correctly implemented. The main security goals of CHUNKED-CACHE are to prevent an adversary from accessing (read/write) data in the exclusive cache chunk of an I-Domain and to prevent eviction interference between the adversary and victim domain. In the following, we show how CHUNKED-CACHE achieves these goals with strict cache partitioning and we discuss why CHUNKED-CACHE's security guarantees even hold in the event of a strong adversary that compromised the operating system kernel. Besides these security considerations, we verified the correctness of our implemented CHUNKED-CACHE prototype by explicitly issuing memory requests which try to read, write and evict cached data of I-Domains.

Strict Partitioning of I-Domain Cache Chunks. As described in Section IV, the trusted software component communicates the number of chunk sets which should be assigned to an I-Domain to the CHUNKED-CACHE cache controller which configures the DCAT and verifies that each cache chunk set is only assigned to a single I-Domain. At every cache memory access, the cache controller uses the domain ID to index the DCAT and to retrieve the list of assigned sets (SID-VEC). Since the assignment of domain IDs and configuration of the DCAT can only be performed by the trusted software component, the indexing logic of the cache controller will never return a cache set which does not belong to the issuer of the memory request. Thus, an adversary is never able to read an I-Domain's exclusive sets (*cache chunk*), write to them or evict them. As a result, CHUNKED-CACHE protects from access-based attacks, which require the adversary to flush memory out of the victim's sets, and conflict-based attacks, which require to fill the victim's sets and thus, evicting its cache lines. Moreover, CHUNKED-CACHE's strict cache resource separation prevents an adversary from observing evictions of its own sets caused by the victim, which protects from occupancy-based attacks, and also strictly prevents the sharing of replacement policy metadata, which has been shown exploitable [51]. In general, the adversary can only infer how many cache sets are assigned to an I-Domain but cannot infer which sets (and therefore which memory addresses) are accessed at which point in time. As described in Section III-B, collision-based attacks are not considered. Defending against them architecturally requires locking the victim cache lines. CHUNKED-CACHE could be extended to integrate this, though mitigating an attack which is very specific to particular software implementations and is not widely shown does not justify the resulting large performance overhead.

CHUNKED-CACHE allows for a dynamic assignment of cache sets to I-Domains. Whenever the *cache chunk* capacity of an I-Domain is modified, all assigned chunk sets are invalidated. This prevents leakage of sensitive I-Domain data when chunk sets are reassigned to another execution domain, and prevents an adversary from injecting malicious data into a set, when additional sets are assigned to an I-Domain. The invalidation is however only required for the I-Domain whose *cache chunk* is resized; all other I-Domains do not need to be modified and thus, their cache lines do not need to be flushed. The same applies when the *cache chunk* for an I-Domain is completely de-allocated. An adversary could also try to trick an I-Domain into storing sensitive data in a mainstream cache line that is accessible for the adversary (SHARED flag bit set). CHUNKED-CACHE prevents this by checking the metadata on every memory request of an I-Domain to verify that the memory region was indeed configured as *shared*.

Protecting from Compromised NI-Domain. As described in Section III-B, in the adversary model of TEE architectures, the OS (and therefore the NI-Domain) is not trusted, allowing an adversary to map physical memory pages of a victim I-Domain to its own memory space and to directly access it in the cache. If an I-Domain (represented by an enclave) demands side-channel protection (EXCLUSIVE-CACHE MODE), all data is cached in the exclusive *cache chunk* and thus, not accessible for the adversary. However, if an I-Domain is not concerned about cache side channels (MAINSTREAM-CACHE MODE), the data is cached in the shared mainstream sets and thus, must still be protected from malicious direct accesses. CHUNKED-CACHE prevents those attacks with the domain ID tag which is added to every cache line. On every cache write, the domain ID tag is set to the ID of the write request issuer. Subsequently, on every read request, the ID of the issuer is compared to the stored ID and the request only permitted if both IDs match. Evictions are permitted for every domain to achieve a perfect utilization of the shared cache sets. This is, however, not a security concern since an I-Domain's data will only be cached in the shared sets if the I-Domain is in MAINSTREAM-CACHE MODE or if the data is explicitly shared with the NI-Domain.

VI. Implementation & Evaluation

To evaluate Chunked-Cache with respect to its hardware footprint, power consumption overheads, and performance impact, we implemented our design in hardware and on an architectural cycle-accurate simulator.

Methodology. We implemented a hardware RTL model of Chunked-Cache to extend an open-source RISC-V processor and synthesized it to evaluate the storage and logic overhead incurred. We use our hardware implementation to extract the additional cycle latencies incurred by Chunked-Cache due to individual cache management and access operations. Then, to evaluate the performance impact of Chunked-Cache on large mixed workloads, we extend an architectural cycle-accurate simulator, the gem5 simulator, with Chunked-Cache and configure it to model a multi-core architecture with a 3-level cache hierarchy which matches our system assumptions (Section III-A). We incorporate the cycle latencies derived from our hardware implementation into our gem5 setup and use it to collect performance measurements on the standard SPEC CPU2017 [20] benchmarks suite (aligned with related work [77], [78], [96], [87]) to evaluate the overall performance impact of Chunked-Cache. Complementary to the compute-intensive SPEC benchmarks, we also evaluate Chunked-Cache on the I/O-intensive web server `nginx`. In order to achieve the most realistic results, we conduct our experiments in the full-system simulation mode of gem5 which simulates the user- and kernel-space software and also I/O devices.

We describe next our hardware implementation (Section VI-A), performance evaluation (Section VI-B), and our hardware an power overhead evaluation (Section VI-C).

A. Hardware Implementation

In our hardware model, we extended the cache tag store with a 4-bit `DID` and a 1-bit `SHARED` bit to tag the owner domain of each cache line and whether it is shared with the NI-Domain (OS), respectively. We also extended the cache controller with the table structures shown in Figure 5. To track the status of the 16,384 sets of a 16 MB LLC with 16-ways, the CST is implemented as a 16,384-bit register that is indexed by the set ID to read out the corresponding 1-bit ALLOC flag. To support set allocation for 16 domains in parallel, the DCAT is implemented as a 16-row `DID`-indexed vector structure. We decided for 16 parallel domains in our hardware implementation since this is also the maximum number of enclaves supported by multiple TEE architectures in parallel [57], [6]. We define for our implementation that the maximum number of sets that can be allocated to any domain is 8,192 sets. Thus, we reserve 4 bits to represent the set INDEX bits number (to index into one of 8,192 sets), 114,688 bits (8,192 sets × 14 bits to represent each set's global ID) for the SID-VEC, and 1 bit ALLOC flag per domain. We discuss the storage overheads incurred by the tables in Section VI-C.

We implement the control finite-state-machines (FSMs) that receive cache allocation and de-allocation requests and perform the necessary management. For allocation, the FSM controls cycling through the sets sequentially to allocate free ones to the requesting I-Domain, updating their status in the CST and updating the corresponding domain status in the DCAT. For de-allocation, another FSM controls that the SID-VEC of the pertinent I-Domain is read from the DCAT, its ALLOC flag reset, and then, all sets of that I-Domain de-allocated (by sequentially indexing through the CST with the respective set IDs from the SID-VEC). Both allocation and de-allocation occur in powers-of-2 set numbers in our prototype. This is only an implementation decision in our prototype to minimize the logic complexity and overhead.

The cache access mechanisms are extended to include the DCAT lookup required for Chunked-Cache to identify which global set IDs belong to the issuing domain and to map the access to the correct set prior to tag lookup. Additionally, for NI-Domain accesses, after mapping to the correct set ID, concurrent sets are computed and looked up in the CST in parallel to identify which ones are unallocated.

B. Performance Evaluation

In this section, we first describe the latencies from our RTL model which we incorporate into our gem5 implementation. Next, we provide an evaluation of Chunked-Cache's performance impact using the gem5 implementation.

Cycle Latencies. As described in Section IV, Chunked-Cache introduces a new indexing policy. For I-Domain memory requests in Exclusive-Cache Mode, a lookup in the DCAT is required. For requests in Mainstream-Cache Mode and all NI-Domain (OS) memory requests, the mainstream sets must be looked up. The comparison of the stored `DID` with the requester `DID` is done in parallel with the address tag comparison and thus, does not introduce additional latency. For I-Domain requests in Exclusive-Cache Mode, we measure an additional latency of 1 cycle and for NI-Domain requests and I-Domain requests in Mainstream-Cache Mode of an additional 2 cycles. For the access latencies of modern LLCs on multi-core systems, we estimate a baseline of 80 cycles in line with vendor multi-core processors [2].

Whenever an I-Domain gets sets allocated, unallocated sets are looked up and the DCAT updated. At de-allocation, sets of the I-Domain must be invalidated (and possibly flushed) and the CST and DCAT updated. For allocation, the overall latency incurred is variable and is a function of: 1.) how many sets CH-NUM are requested for allocation, and 2.) how many sets have to be looked up in the CST. At the worst case, this incurs a latency of 16,384 cycles and at the best case, CH-NUM cycles. An additional 1 cycle is incurred to update the DCAT subsequently. The INDEX is computed and communicated already by the trusted component in the allocation request, thus it does not contribute additional latency.

For de-allocation, we measure an overall latency of CH-NUM + 2 cycles, where 1 cycle is required to look up the DCAT, and another cycle to update it, followed by CH-NUM cycles to de-allocate each set in the CST. At worst case, a latency of 8,194 cycles is incurred (assuming a maximum of 8,192 sets per domain). However, de-allocating the sets in CST is done in parallel to invalidating (and possibly flushing if dirty) the respective cache lines.

We emphasize that allocating new sets to any I-Domain does not require invalidating or flushing any other sets of

Parameters	L1 (I&D)	L2	L3 (gem5)	L3 (CHUNKED-CACHE)
size	64 KB & 32 KB	512 KB	16 MB	16 MB
# of sets	128 & 64	512	16,384	16,384
associativity	8-way	16-way	16-way	16-way
access latency (in cycles)	4	14	80	81 / 82

TABLE I. CACHE CONFIGURATION ON OUR GEM5 EVALUATION SETUP WITH AN INCLUSIVE 3-LEVEL CACHE HIERARCHY.

the NI-Domain or other I-Domains which would require re-caching them. This is one key design goal of CHUNKED-CACHE since it eliminates this performance overhead on other domains, particularly the NI-Domain. The allocation of sets either happens only once during the I-Domain setup or occasionally when the number of assigned sets is modified at run-time which requires a context switch out of the I-Domain. The CHUNKED-CACHE allocation/de-allocation overheads induced remain negligible when compared with the general overheads of TEE architectures [22], [13], [57], [6]. Therefore, we do not invest in increased logic complexity to optimize the cycle overheads incurred for allocation and de-allocation, since they are not in the critical path, i.e., LLC accesses.

Mixed-Workload Cycle-Accurate Evaluation. We implement CHUNKED-CACHE on the cycle-accurate gem5 simulator and construct a multi-core system which resembles a modern computing system with an inclusive 3-level cache hierarchy. Each core has access to a core-exclusive L1 and L2 cache, and an L3 LLC shared among all cores. For the L1 and L2, we use the unmodified cache implementation provided by gem5, whereas we use our CHUNKED-CACHE implementation for the L3 cache. The configuration parameters of each cache level are shown in Table I. We derive realistic values for the cache sizes, number of cache sets, associativity and access latency in line with modern caches. For the CHUNKED-CACHE L3 cache, we add our induced latencies collected from our hardware implementation. Constructing a gem5-based multi-core system with 3-level cache hierarchy in full-system simulation mode to collect representative cycle-accurate traces for large workloads involved significant engineering challenges, as also evident by recent works that rely on trace-based simulators for their evaluation with SPEC workloads [77], [78], [96], [87].

We measure the performance impact of CHUNKED-CACHE on real-world workloads by using the standard SPEC CPU2017 benchmarks with both the SPECspeed 2017 Integer and SPECspeed 2017 Floating Point suites which represent a wide range of compute-intensive applications such as compilers, video compression, machine learning or modeling tasks. Since running all of the benchmarks on our full-system cycle-accurate gem5-based simulation setup would be very costly in terms of memory and time, we selected benchmarks from the different application domains and different working set sizes, guided by this memory-centric characterization of the SPEC CPU2017 benchmarks [85]. Moreover, to also cover I/O-intensive workloads, we evaluate the impact of CHUNKED-CACHE on the widely used web server `nginx`. We run our experiments for 1 trillion instructions before we start to collect measurements, in order to boot the system, start the benchmarks and collect more representative metrics. We run all our experiments for a total of 1 billion instructions in the full-system mode of gem5 and collect statistics to compute the Cycles Per Instruction (CPI) metric, in order to capture the additional latency effect, and the L3 cache miss rates for the reduced cache capacity effects. If not stated otherwise for single experiments, the miss rates are calculated as the geometric mean over the instruction and data miss rates of the page table walker and core. We compare CHUNKED-CACHE to 1.) a baseline system with an unmodified insecure L3 cache and to 2.) an L3 cache which implements a way-based partitioning scheme in which cache ways are assigned to I-Domains as provided, e.g., by CATalyst [61] which uses Intel CAT [49], SecDCP [94], DAWG [51], Keystone [57] or CURE [6]. We evaluate CHUNKED-CACHE with a set of experiments which investigate different computing scenarios. First, we show how CHUNKED-CACHE's partitioning influences the performance of mixed workloads when encapsulated in I-Domains (in EXCLUSIVE-CACHE MODE). Then, we evaluate CHUNKED-CACHE's impact on the NI-Domain (OS-domain) and compare against way-based partitioned cache schemes. We conclude our evaluation with a set of experiments which show the scalability of CHUNKED-CACHE. In general, when comparing to the baseline (unpartitioned L3 cache shared by the same workload), our experiments show a negative effect of CHUNKED-CACHE on the performance of a benchmark when only a small *cache chunk* size is assigned to it. However, when increasing the *cache chunk* size this effect vanishes. At some point, depending on the specific characteristics of a benchmark, the exclusive *cache chunk* assigned by CHUNKED-CACHE leads to a positive effect on its performance as we show in the following experiments. This gives the developer some degree of freedom to calibrate the performance of the workload by distributing the cache resources accordingly, e.g., to optimize the performance of a particular benchmark if desired given that the cache resources are available/affordable. All experiments were conducted on an x86 platform equipped with an Intel Xeon Silver 4215 CPU (2.50 GHz) and 186 GB RAM.

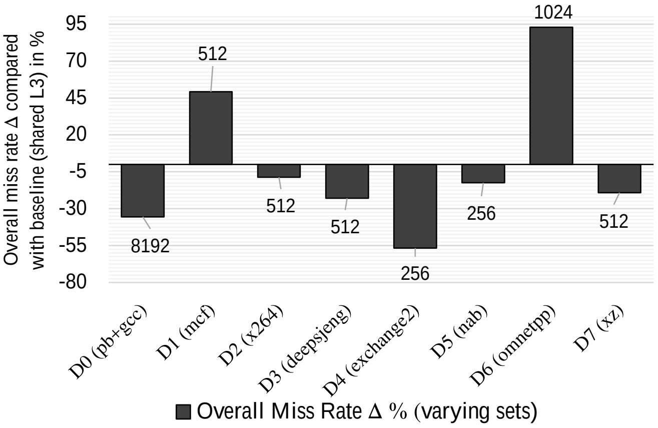


Fig. 6. Cache miss rate impact of CHUNKED-CACHE for SPEC benchmarks on a 8-domain setup; compared to a shared L3 cache.

I-Domain Performance Impact. In the first set of experiments, we evaluate the performance impact CHUNKED-CACHE has on mixed workloads when protected in I-Domains in EXCLUSIVE-CACHE MODE. We run 7 randomly selected SPEC benchmarks in I-Domains and show our re-

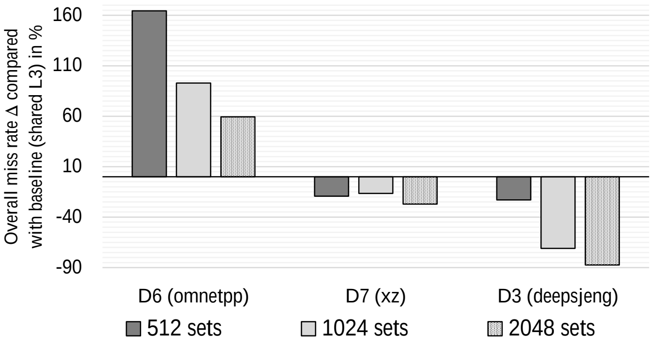


Fig. 7. Cache miss rate impact of Chunked-Cache for SPEC CPU2017 benchmarks (varying sets); compared to a shared L3 cache.

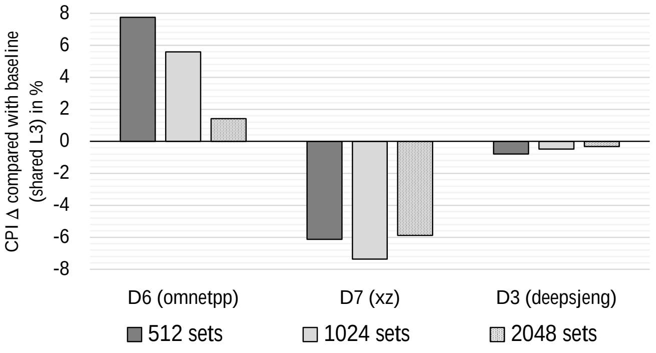


Fig. 8. CPI impact of Chunked-Cache for SPEC CPU2017 benchmarks (increasing sets); compared to a shared L3 cache.

sults in Figure 6. The NI-Domain (D0) runs Linux (kernel version 4.19.83) and 2 benchmarks with large working sets (`600.perlbench_s` and `602.gcc_s`). In this experiment, we assign 8,192 sets to the NI-Domain and a varying number of sets to each I-Domain as indicated in the plot. We chose the number of sets by briefly analyzing the working set size of the benchmark running in each I-Domain, and assigning bigger working sets to more cache sets. This is only required when optimizing for performance, otherwise a default number of sets can be assigned to each benchmark. We observe in the experiment that the overall miss rate significantly decreases for most benchmarks when compared to sharing the L3 cache. This shows that the assignment of a smaller but exclusive cache portion can even reduce the cache miss rates of a workload. Moreover, our results indicate that the number of cache sets required to reduce or completely avoid the impact of Chunked-Cache heavily depends on the characteristics of the workload. In our experiment, the benchmarks `605.mcf_s` and `620.omnetpp_s` would require more cache sets than the assigned 512 and 1024 sets to avoid an impact on the cache miss rates. We investigate this in another experiment where we customize the number of sets allocated to an I-Domain for some of the benchmarks and show how the miss rate decreases significantly when increasing the chunk size (Figure 7). In another experiment (Figure 8), we show how the varying chunk sizes also influence the CPI values. As for the miss rates, the CPI decreases in general. We observe, however, some outliers with the CPI metrics collected, owing to the complexity of a full-system multi-core simulation on gem5 which also includes unpredictable kernel runtime behavior into the statistics.

Additionally, to evaluate the impact of Chunked-Cache on I/O-intensive workloads, we conduct experiments in which we run the `nginx` web server in one I-Domain and the HTTP benchmarking tool `wrk` in another I-Domain, whereas we keep the NI-Domain unmodified. We then use `wrk` to send HTTP requests to the web server using 12 threads and 400 open connections. In Figure 9, the miss rate impact of Chunked-Cache on `nginx` and `wrk` is shown when increasing the number of sets from 128 to 2048. The results show, in line with our results on SPEC, how the increase of cache sets leads to a decrease in the overall miss rate. The decrease is already noticeable for a relatively small number of sets since the exclusive assignment of the cache sets prevents `nginx` and `wrk` from evicting the sets from one another.

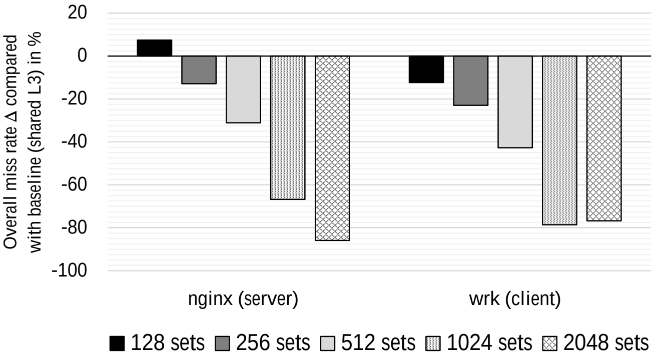


Fig. 9. Cache miss rate impact of Chunked-Cache for `nginx` and `wrk` (increasing sets); compared to a shared L3 cache.

NI-Domain Performance Impact. In the second set of experiments, we focus on the performance impact of Chunked-Cache on workloads executing in the NI-Domain. We again run mixed workloads from the SPEC benchmarks in I-Domains, while running Linux and the 2 memory-intensive benchmarks `600.perlbench_s` and `602.gcc_s` in the NI-Domain. In Figure 10, we vary the number of sets allocated to the NI-Domain from 2,084 to 8,192 while keeping the sets for the other domains unchanged. For these experiments, we show all 4 miss rate metrics over which we average in the other experiments, the data and instruction miss rates of the page table walker (DTB MR and ITB MR, respectively), and the data and instruction miss rates of the core (Data MR and Instr. MR, respectively). While in general, all miss rates and CPI metrics decrease compared to the baseline, we only observe a slight improvement when increasing the chunk size from 2,084 to 4,096 and 8,192 sets. This is because even when the number of statically allocated sets to the NI-Domain is rather small, the unallocated sets in the system (mainstream sets) remain available for the NI-Domain. Thus, performance is not significantly impacted for the NI-Domain and maximum utilization of the available resources (as in an unmodified insecure cache architecture) is preserved which was one of the key design goals of Chunked-Cache.

To investigate this, we run experiments (same setup) in

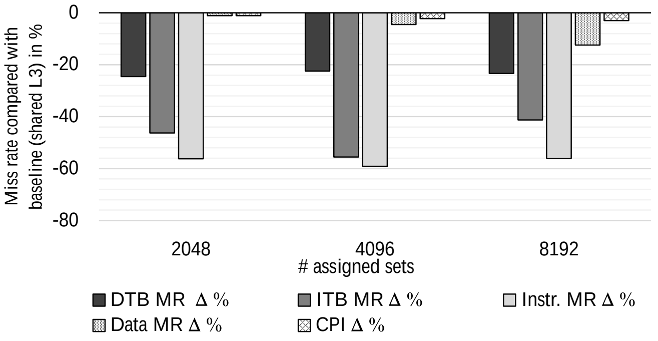


Fig. 10. Miss rate & CPI impact of CHUNKED-CACHE on the NI-Domain (increasing sets); compared to a shared L3 cache.

which we assign 1,024 sets to the NI-Domain and vary the number of unassigned sets. In the first run, all cache sets are allocated in our system, while in the second run, 4,096 sets remain unallocated and available for the NI-Domain. Figure 11 shows how the miss rates significantly decrease when 4,096 sets remain unallocated which demonstrates how CHUNKED-CACHE enables the NI-Domain to utilize unused cache sets.

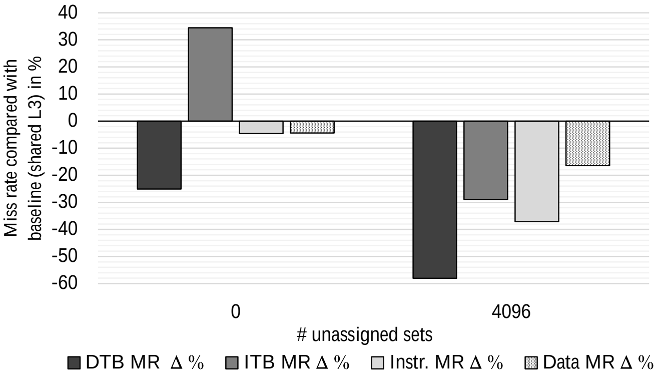


Fig. 11. Miss rate of CHUNKED-CACHE on the NI-Domain with varying number of unassigned sets; compared to a shared L3 cache.

Comparison with Partitioning-based Schemes. We compare CHUNKED-CACHE to a cache partitioning scheme which we implement on gem5, specifically way-based partitioning, being the only other strict cache partitioning approach. We run a number of experiments with a 5-domain setup where we assign the same cache capacity to the same benchmark in both, the CHUNKED-CACHE and way-partitioned cache – 1,024 or 2,048 sets in CHUNKED-CACHE and equivalently 1 way or 2 ways, respectively, in the way-partitioned setup. We show in Figure 12 how for the same cache capacity, CHUNKED-CACHE outperforms way-based partitioning for randomly selected benchmarks. In fact, for some benchmarks such as `625.x264_s` and `644.nab_s`, allocating 1,024 sets even outperforms 2 ways (double the cache capacity) on a way-partitioned cache. We calculate an average decrease of 43% in the miss rate for CHUNKED-CACHE vs. the way-partitioned cache for a 1 MB cache capacity (1024 sets) and a 39% decrease for 2 MB (2048 sets).

Scalability and Dynamic Cache Allocation. In Appendix A, we additionally evaluate CHUNKED-CACHE's ability to scale and support 32 I-Domains in parallel without degrading the performance of the NI-Domain (OS) and we also demonstrate how CHUNKED-CACHE supports the dynamic allocation of cache sets to an I-Domain during runtime.

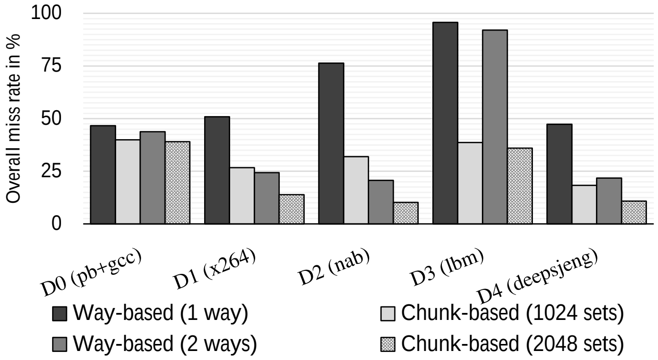


Fig. 12. Overall miss rate for SPEC CPU2017 benchmarks with CHUNKED-CACHE; compared to a way-partitioned cache.

C. *Hardware Footprint and Power Consumption Evaluation*

To evaluate the storage and logic overhead incurred by CHUNKED-CACHE, we synthesize our implementation targeting a single-issue single-core RISC-V processor [79] using Xilinx Vivado tools. While this processor does not provide an LLC, this is not necessary since we can still extend the existing simple cache controller to implement CHUNKED-CACHE, verify its functionality in cycle-accurate RTL-level simulations and evaluate its overheads.

Storage/Memory Overhead. The main contribution to the hardware area overhead of CHUNKED-CACHE is the extra storage required, rather than the logic itself, since that requires the fabrication of memory which consumes more gates than hardware logic. The extra storage is needed for the additional tag bits required per cache line (4-bit `DID` and 1-bit `SHARED` flag), the CST and DCAT. In our current prototype implementation targeting 16 domains, 16 MB LLC with 16-ways and 16,384 sets, and an allowed maximum of 8,192 sets per domain, the CST consumes 2 KB, the DCAT ≈ 224 KB, and the additional tag storage 160 KB, totaling 386 KB. This amounts to a negligible 2.3% storage overhead relative to a 16 MB LLC which would consume approximately an additional 2.7% area in fabrication.

The capacity of these tables and the consequent storage area overheads are directly impacted by how the various design/implementation trade-offs involved are configured in different implementations of CHUNKED-CACHE, namely 1) the number of active parallel domains supported (overhead increases only linearly) 2.) the total L3 cache capacity and its number of sets, and 3.) the maximum number of sets that can be allocated to a domain. For example, to support 32 domains, one more tag store bit is required costing an additional 0.25KB, relative to the overhead incurred for 16 domains as described above. The CST capacity is unaffected, while the DCAT capacity doubles to 448KB. The power consumption (evaluated below) would increase proportionally.

Logic Overhead. CHUNKED-CACHE requires extra hardware logic for the FSMs that handle the cache de-/allocation, and look up the tables prior to cache accesses (Section VI-A). We synthesize our hardware implementation using Xilinx Vivado targeting a ZedBoard Zynq-7000 FPGA board, and estimate a logic overhead of $\approx$ 1.6% relative to the single-core RISC-V processor that we extend. This would diminish relative to a significantly more complex multi-billion-transistor processor with a 3-level cache hierarchy which is the intended platform for CHUNKED-CACHE. Furthermore, this overhead does not increase as the number of domains supported by CHUNKED-CACHE increases.

Power Consumption Overhead. We focus here on the power consumption overheads incurred by the extended tag store and CST and DCAT tables, since the extra hardware added is dominated by them, and they contribute the most to the additional power consumption (static and dynamic) overheads. Besides, power consumption by cache memories is significantly more than logic, and is usually the largest contributor to the total power consumed by a chip. We estimate the power consumption overheads of CHUNKED-CACHE in 22nm technology using the CACTI-6.0 tool [70]. For a 16-way 16 MB cache with 64 B cache line size, the total leakage power increases from 5056.57 mW (baseline) to 5313.83 mW. The CST and DCAT incur an additional 365 mW, amounting to a total of 12.3% increase in the LLC power consumption. To support OS-specific chunk set indexing, the power consumption increases accordingly. If 2 sets are looked up in parallel (when 8,192 sets are allocated to the OS), the penalty on power consumption is negligibly minimal. When 4 or 8 sets are looked up in parallel, the power consumption overhead additionally increases by 5.5% and 27.1% relative to the baseline of 5056.57 mW, respectively. Relative to the overall chip power consumption of modern multi-core processors (90-150W), the LLC power consumption increases incurred by CHUNKED-CACHE remain reasonable.

VII. RELATED WORK

We categorize cache side-channel defenses which tackle the problem directly in the cache into two broad classes: partitioning-based and randomization-based. We focus in this section only on the most relevant works to CHUNKED-CACHE, which all propose hardware changes at the cache architecture.

A. Partitioning-based Microarchitectures

The partitioning-based defenses most related to CHUNKED-CACHE propose new cache architectures that assign cache resources (cache lines or ways) exclusively to protected domains. The TEE architectures Keystone [57] and CURE [6] implement way-based partitioning to assign cache ways exclusively to enclaves. SecDCP [94] forms security classes of applications with similar security requirements and assigns cache ways to them. DAWG [51] provides way-based cache partitioning in the context of speculative execution attacks. The main limitation of way-based partitioning is its inability to support a large number of protected domains in parallel since even large LLCs only comprise a small number of cache ways (up to 16). Moreover, these defenses lead to cache underutilization when assigned cache ways are not evenly utilized by a protected domain since the unused cache lines are blocked for all other domains on the system.

CHUNKED-CACHE, besides other approaches [95], [23], is more flexible since it partitions the cache on a cache-line basis. PLcache [95] assigns cache lines exclusively to processes which allows for a strict and fine-grained partitioning of cache resources. However, PLcache's strict isolation does not allow for caching data shared between processes and strongly impacts the overall system performance and fairness of the cache utilization. Moreover, PLcache does not protect against occupancy-based attacks since the adversary can still infer the victim's memory accesses by observing that the victim is unable to access/evict cache lines.

HybCache [23] assigns cache ways to protected domains (or enclaves) by providing a fully-associative mapping with random replacement for the ways to overcome the cache underutilization problem of way-based partitioning schemes. In contrast to PLcache, HybCache assigns only a subset of the cache resources to the protected domains which can be reclaimed by non-sensitive domains and thus, a fairer cache utilization is achieved which does not heavily degrade the overall system performance. However, HybCache does not scale practically with large LLCs since it would incur high power consumption overheads. Moreover, HybCache does not provide strong security guarantees against occupancy-based attacks since it does not enforce a strict partitioning.

In memory page-coloring schemes [29], [50], [101], [61], [22], the mapping from physical memory addresses to cache lines is utilized to ensure that the cache lines used from sensitive applications do not overlap. One problem with page-coloring is its high impact on the software memory layout. It cannot fully support DMA and requires modifying the memory management (OS or hypervisor). Moreover, the assignment of cache lines is static, i.e., modifying the number of assigned cache lines during runtime would require to alter the physical memory layout of the software which is highly impractical.

CHUNKED-CACHE, however, provides flexible cache-line partitioning that can scale to support a larger number of protection domains than the number of cache ways. It additionally overcomes the limitations of other cache-line partitioning techniques by providing support for shared memory and by scaling to large LLCs while still providing strict isolation. In contrast to page coloring schemes, CHUNKED-CACHE does not influence the memory layout, is compatible with commodity memory management software, and allows dynamic modification of the chunk sizes during runtime.

B. Cryptographic Randomization Defenses

These randomization techniques attempt to avoid the storage overhead of large randomized mapping tables that are deployed by earlier defenses [95], [63], [62] by relying on cryptographic primitives to reproducibly generate the randomized mapping. Time-Secure Cache [89] uses a set-associative cache indexed with a keyed function using the cache line address and process ID as its input. However, a weak low-entropy indexing function is used, thus, frequent re-keying and cache flushing must be performed which increases complexity and performance impact.

CEASER [77] also uses a keyed indexing function but without process ID. It also requires frequent re-keying of its index derivation function and re-mapping to limit the time interval available for an attacker to reconstruct the eviction set. Under a minimal eviction set construction algorithm of $\mathcal{O}(E^2)$ complexity, CEASER has been shown able to withstand attacks with a re-keying rate of 1%. However, under eviction set construction techniques with $\mathcal{O}(E)$ complexity [78], the re-keying rate needs to increase to 35%-100%, which incurs prohibitively high performance overheads. To resist these improved attacks, a skewed variant of CEASER, CEASER-S [78] was proposed that divides the cache ways into multiple partitions (skews), with different encryption keys used for each partition. A cache line maps to a different set in each partition, where one of the partitions is chosen randomly for the line placement, making the minimal eviction set construction more difficult.

ScatterCache [96] also uses keyed cryptographic indexing where cache set indexing is different and pseudo-random for every protected domain but consistent for any given key. Thus, re-keying is still required at time intervals to hinder the profiling and minimal eviction set construction efforts.

Phantom-Cache [87] relies on a set of hardware-efficient hash function and XOR operations to map a cache line to 1 of 8 randomly chosen sets in the cache, each with 16 ways, thus, increasing the associativity to 128. This requires accessing 128 locations on each cache access to check if an address is cached, resulting in a high power overhead of 67%.

Defenses based on cryptographic primitives have multiple weaknesses: 1.) These defenses remain only as secure as the best/fastest known attack strategy/minimal set eviction construction algorithm [12], [75] with no solid future-proof security guarantees. In fact, a recent work [86] has further shown that other attack techniques and workarounds can be used to exploit certain flaws in ScatterCache and CEASER-S to completely undermine them and their security guarantees. 2.) Their promised security guarantees often rely on the alleged, yet not thoroughly investigated unpredictability of low-latency cryptographic primitives. The primitives deployed by CEASER, CEASER-S and ScatterCache have been shown vulnerable to cryptanalysis which enables the construction of eviction sets without even accessing memory [74], [10]. Deploying primitives that resist formal cryptanalysis is also not practical since it would incur increased latency, thus, further degrading performance in the cache's critical path. 3.) If the re-keying rate is increased to mitigate novel attacks, the induced performance overhead renders these defenses impractical.

Mirage, a concurrent work, attempts to overcome the vulnerability to newer faster eviction-set construction algorithms, by eliminating set-associative eviction altogether [80]. However, besides still being vulnerable to occupancy-based attacks, Mirage does not support selectively enabling side-channel resilience only for execution domains that require it, thus, incurring a performance slowdown on the entire workload.

CHUNKED-CACHE, in contrast, eliminates the described unreliability and inflexibility fundamentally by providing strict, yet perfectly configurable and selective, partitioning across the execution domains. This enables each domain to allocate the cache capacity it requires and thus, experience the performance that it has opted to tolerate accordingly. This different paradigm provides well-grounded security assurances that stand the test of advances in cache side-channel attacks and different attack methodologies and complexities, without sacrificing performance. Instead, it provides by-design the possibility to tune the security-performance trade-off for each domain as desired, without overtaxing the OS either.

VIII. CONCLUSION

In this paper, we presented a novel side-channel-resilient cache microarchitecture, CHUNKED-CACHE, for TEE architectures, that enables each execution domain to flexibly and selectively configure its exclusive cache sets only when cache isolation and side-channel resilience is required. Unlike randomization-based cache microarchitectures recently proposed, CHUNKED-CACHE fundamentally mitigates side-channel attacks by enforcing strict cache partitioning, thus providing future-proof and solid security guarantees. It also outperforms way-based partitioning and scales to support a larger number of execution domains, without degrading the performance of the OS. In this work, we show how CHUNKED-CACHE incorporates this configurable performance-security trade-off by design in the cache microarchitecture to cater most optimally for TEE architectures. Through our security analysis and evaluation, we also show how on-demand sophisticated side-channel security, as well as performance, functionality and usability requirements are preserved in CHUNKED-CACHE, with small hardware and memory costs.

ACKNOWLEDGEMENT

We thank our anonymous reviewers for their valuable and constructive feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – CROSSING/236615297.

APPENDIX

A. I-Domain Scalability

We also demonstrate how CHUNKED-CACHE scales for a larger number of parallel domains. As described in Section IV, the design of CHUNKED-CACHE allows to support more domains in parallel than the 16 domains we choose for our hardware implementation. Thus, we conduct scaling experiments where we run the `619.lbm_s` benchmark on every I-Domain and we increase the number of I-Domains from 4 to 8, 16 and up to 32. Running more I-Domains in parallel is not possible on our evaluation platform since the gem5 full-system simulation with 32 I-Domains already consumes the complete 186 GB of available RAM which unavoidably imposes certain limitations on our experiments. Given these constraints, we selected `619.lbm_s` as a benchmark because of its relatively small working set. Throughout these experiments, the NI-Domain (which runs the Linux kernel and one instance of `619.lbm_s`) gets 8,192 sets assigned. The overall miss rates for the NI-Domain, when scaling from 4 to 32 I-Domains, are stable, reaching 71.45%, 71.64%, 72.06% and 71.75%, respectively. Thus, with CHUNKED-CACHE, also a high number of I-Domains can be supported without degrading the performance of the NI-Domain (OS). Running even more domains was only limited by the memory constraints of our evaluation platform.

B. Dynamic Set Allocation

In another experiment, we analyze how the dynamic set allocation capabilities of CHUNKED-CACHE impact the NI-Domain and I-Domains during runtime. For this, we select a SPEC benchmark (631.deepsjeng_s) which achieves a relatively small average cache miss rate, when enough cache sets are available, in order to better demonstrate the behavior of the dynamic set allocation. We run the benchmark in 4 distinct I-Domains and as part of the NI-Domain. We simulate 24 billion cycles on our evaluation platform which corresponds to 12s worth of computing (given that we simulate processors with a clock frequency of 2 GHz). At the beginning of the experiment, the NI-Domain (D0) gets 8,192 sets assigned, the I-Domains D1-D3 512 sets each and the I-Domain D4 only 1 set. Then, during runtime, the size of D4's chunk is modified. After 3s, the chunk size is increased to 512 sets, after 6s to 2048 sets and after 9s decreased to 1 set. The chunk sizes of the domains D0, D1, D2 and D3 are kept constant throughout the duration of the experiment. We collect miss rate statistics for all domains every 75ms (150,000,000 cycles) and compute the arithmetic mean over the instruction and data miss rates of the page table walker and core.

The results of the experiment are shown in Figure 13, whereby we only show the miss rates for D0, D1 and D4 since the results of D2 and D3 are very similar to those of D1. The plot clearly shows how the increase and decrease of the chunks size affects the miss rate of D4. At the beginning, when only 1 set is assigned to D4, the miss rate fluctuates heavily around a value of 80%. At the time point 3s, when 511 additional sets are assigned to D4, the miss rate almost immediately drops to around 60%, thereby catching up with the miss rates achieved by D1. After another 3s, when D4's chunk size is increased to 2048, a low and stable miss rate of 20% is achieved. The fact that D0 experiences the same miss rate with 8,192 sets shows that applications are not always benefiting from an increased chunk size and thus, available sets are better redistributed to other benefiting domains to improve the overall system performance. After 9s, the chunk size is decreased to 1 set which again leads to a heavily fluctuating miss rate of around 80%.

Another interesting take-way from Figure 13 is that the flushing of all chunk sets, which happens after 6s, does not negatively influence the miss rate of D4, at least not when collecting the miss rate statistics at intervals of 75ms.

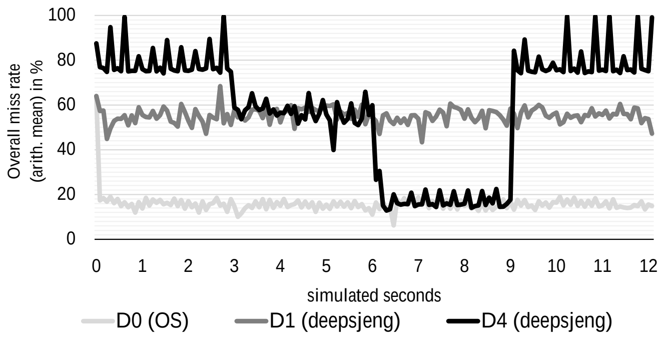


Fig. 13. Cache miss rate impact of CHUNKED-CACHE on the NI-Domain (D0) and I-Domains (D1, D4) when dynamically modifying the size of the cache chunk assigned to D1.

REFERENCES

[1] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[2] Intel Skylake X. https://www.7-cpu.com/cpu/Skylake_X.html, 2020.

[3] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. *ACM Symposium on Information, computer and communications security*, pages 312–320, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. *Cryptographers' Track at the RSA Conference*, pages 225–242, 2007.

[5] 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, 2009.

[6] 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 *USENIX Security Symposium*, 2021.

[7] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[8] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, pages 159–176. Springer, 2012.

[9] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystem. In *CRYPTO*, 2000.

[10] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro. Brutus: Refuting the Security Claims of the Cache Timing Randomization Countermeasure Proposed in CEASER. *IEEE Computer Architecture Letters*, 2020.

[11] Joseph Bonneau and Ilya Mironov. Cache-collision Timing Attacks Against AES. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2006.

[12] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *Micro*, 2020.

[13] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[14] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2017.

[15] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *IEEE Symposium on Security and Privacy*, 2020.

[16] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[17] 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 *IEEE European Symposium on Security and Privacy*, 2019.

[18] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing*, 49:1162–1174, 2016.

[19] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In *CCS*, 2014.

[20] Standard Performance Evaluation Corporation. SPEC CPU 2017. https://www.spec.org/cpu2017, 2017.

[21] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical report, Cryptology ePrint Archive. Report 2016/086, 2016. https://eprint.iacr.org/2016/086.pdf.

[22] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium*, 2016.

[23] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security Symposium*, 2020.

[24] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, 2017.

[25] Goran Doychev and Boris Köpf. Rigorous Analysis of Software Countermeasures Against Cache Attacks. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2017.

[26] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*. ACM, 2013.

[27] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[28] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, Dmitry Ponomarev, et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 693–707, 2018.

[29] Michael Godfrey. On The Prevention of Cache-Based Side-Channel Attacks in a Cloud Environment. Master's thesis, Queen's University, Ontario, Canada, 2013.

[30] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, 2017.

[31] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*, 2018.

[32] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[33] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory. In *USENIX Security Symposium*. USENIX Association, 2017.

[34] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[35] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer-Verlag, 2016.

[36] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[37] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2016.

[38] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2011.

[39] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[40] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[41] Intel. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[42] Intel. Intel Integrated Performance Primitives Cryptography Developer Reference. 2019.

[43] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[44] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2016.

[45] Kaplan et al. AMD memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[46] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A High-resolution Side-channel Attack on Last-level Cache. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[47] Mehmet Kayaalp, Khaled N. Khasawneh, Hodjat Asghari Esfeden, Jesse Elwell, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. RIC: Relaxed Inclusion Caches for mitigating LLC side-channel attacks. In *IEEE/ACM Design Automation Conference (DAC)*, 2017.

[48] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[49] Khang T Nguyen. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/articles/introduction-to-cache-allocation-technology, 2016.

[50] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*. USENIX Association, 2012.

[51] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[52] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[53] Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, 1996.

[54] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[55] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-channels. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2012.

[56] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Security Symposium*, 2018.

[57] 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 (EuroSys)*. Association for Computing Machinery, 2020.

[58] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *USENIX Security Symposium*, pages 16–18, 2017.

[59] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[60] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, 2018.

[61] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating Last-Level

Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[62] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

[63] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[64] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[65] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[66] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 118–129. IEEE, 2012.

[67] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2012.

[68] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. *Cryptographers' Track at the RSA Conference*, pages 21–44, 2018. 10.1007/978-3-319-76953-0_2.

[69] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[70] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. https://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf, 2009.

[71] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[72] Mathias Payer. Hexpads: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[73] Colin Percival. Cache missing for fun and profit, 2005.

[74] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic Analysis of Randomization-based Protected Cache Architectures. In *IEEE Symposium on Security and Privacy*, 2021.

[75] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache. *arXiv preprint arXiv:1908.03383*, 2019.

[76] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 195–209, 2019.

[77] Moinuddin K. Qureshi. Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[78] Moinuddin K. Qureshi. New Attacks and Defense for Encrypted-Address Cache. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2019.

[79] Roa Logic BV. RV12. https://github.com/RoaLogic/RV12, 2020.

[80] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX Security Symposium*, 2021.

[81] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 2019.

[82] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[83] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[84] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 639–656, 2019.

[85] Sarabjeet Singh and Manu Awasthi. Memory Centric Characterization and Analysis of SPEC CPU2017 Suite. *arXiv preprint arXiv:1910.00651*, 2019.

[86] Wei Song, Boya Li, Zihan Xue, Zhenzhen Li, Wenhao Wang, and Peng Liu. Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2021.

[87] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *NDSS*, 2020.

[88] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, 2017.

[89] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2018.

[90] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[91] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security Symposium*, 2018.

[92] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *IEEE Symposium on Security and Privacy*, 2019.

[93] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 41–46, 2011.

[94] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2016.

[95] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2007.

[96] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[97] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Atacks. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2017.

[98] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. To appear in the *Proceedings of the IEEE Symposium on Security & Privacy (IEEE S&P)*, May 2019.

[99] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[100] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. volume 7, pages 99–112. Springer, 2017.

[101] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y. Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. Cryptology ePrint Archive, Report 2016/980, 2016. https://eprint.iacr.org/2016/980.

[102] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[103] 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*, pages 1723–1740, 2019.