FLEXIBLE HARDWARE-BASED SECURITY-AWARE MECHANISMS AND ARCHITECTURES

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

MSc. Ghada Dessouky

Referenten:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (Erstreferent)
Prof. Dr. Farinaz Koushanfar (Zweitreferent)

Tag der Einreichung: 23. August 2021
Tag der Disputation: 06. October 2021

TECHNISCHE
UNIVERSITÄT
DARMSTADT

System Security Lab
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

Ghada Dessouky:
Flexible Hardware-Based Security-Aware Mechanisms and Architectures, © August 2021

PHD REFEREES:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (1st PhD Referee)
Prof. Dr. Farinaz Koushanfar (2nd PhD Referee)

PHD DEFENSE COMMITTEE MEMBERS:
Prof. Dr. Iryna Gurevych
Prof. Dr. Christian Reuter
Prof. Dr. Thomas Schneider

First submission 23 August 2021
Dissertation defense 6 October 2021

Dissertation publication at TUprints 2023
URN of dissertation: urn:nbn:de:tuda-tuprints-230426

Erklärung gemäß §9 der Promotionsordnung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwednung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Germany, August 2021

Ghada Dessouky

ABSTRACT

For decades, software security has been the primary focus in securing our computing platforms. Hardware was always assumed trusted, and inherently served as the foundation, and thus the root of trust, of our systems. This has been further leveraged in developing hardware-based dedicated security extensions and architectures to protect software from attacks exploiting software vulnerabilities such as memory corruption. However, the recent outbreak of microarchitectural attacks has shaken these long-established trust assumptions in hardware entirely, thereby threatening the security of all of our computing platforms and bringing hardware and microarchitectural security under scrutiny. These attacks have undeniably revealed the grave consequences of hardware/microarchitecture security flaws to the entire platform security, and how they can even subvert the security guarantees promised by dedicated security architectures. Furthermore, they shed light on the sophisticated challenges particular to hardware/microarchitectural security; it is more critical (and more challenging) to extensively analyze the hardware for security flaws prior to production, since hardware, unlike software, cannot be patched/updated once fabricated.

Hardware cannot reliably serve as the root of trust anymore, unless we develop and adopt new design paradigms where security is proactively addressed and scrutinized across the full stack of our computing platforms, at all hardware design and implementation layers. Furthermore, novel flexible security-aware design mechanisms are required to be incorporated in processor microarchitecture and hardware-assisted security architectures, that can practically address the inherent conflict between performance and security by allowing that the trade-off is configured to adapt to the desired requirements.

In this thesis, we investigate the prospects and implications at the intersection of hardware and security that emerge across the full stack of our computing platforms and System-on-Chips (SoCs). On one front, we investigate how we can leverage hardware and its advantages, in contrast to software, to build more efficient and effective security extensions that serve security architectures, e.g., by providing execution attestation and enforcement, to protect the software from attacks exploiting software vulnerabilities. We further propose that they are microarchitecturally configured at runtime to provide different types of security services, thus adapting flexibly to different deployment requirements. On another front, we investigate how we can protect these hardware-assisted security architectures and extensions themselves from microarchitectural and software attacks that exploit design flaws that originate in the hardware, e.g., insecure resource sharing in SoCs. More particularly, we focus in this thesis on cache-based side-channel attacks, where we propose sophisticated cache designs, that fundamentally mitigate these attacks, while still preserving performance by enabling that the performance-security trade-off is configured by design. We also investigate how these can be incorporated into flexible and customizable security architectures, thus complementing them to further support a wide spectrum of emerging applications with different performance/se-

curity requirements. Lastly, we inspect our computing platforms further beneath the design layer, by scrutinizing how the actual implementation of these mechanisms is yet another potential attack surface. We explore how the security of hardware designs and implementations is currently analyzed prior to fabrication, while shedding light on how state-of-the-art hardware security analysis techniques are fundamentally limited, and the potential for improved and scalable approaches.

ZUSAMMENFASSUNG

Jahrzehntelang stand die Softwaresicherheit bei der Sicherung unserer Computerplattformen im Vordergrund. Die Hardware wurde immer als vertrauenswürdig angesehen und diente als Grundlage und somit als Vertrauensbasis für unsere Systeme. Dies wurde bei der Entwicklung von hardwarebasierten Sicherheitserweiterungen und -architekturen weiter genutzt, um Software vor Angriffen zu schützen, die Software-Schwachstellen, wie z.B. Speicher- und Programmierfehler, ausnutzen. Der aktuelle Ausbruch von Angriffen auf die Prozessor-Mikroarchitektur hat jedoch diese seit langem etablierten Annahmen über das Vertrauen in die Hardware völlig erschüttert, wodurch die Sicherheit aller unserer Computerplattformen bedroht ist und die Sicherheit von Hardware und Mikroarchitektur auf den Prüfstand gestellt wird. Diese Angriffe haben unbestreitbar die schwerwiegenden Folgen von Sicherheitsmängeln in der Hardware/Mikroarchitektur für die Sicherheit der gesamten Plattform aufgezeigt. Zudem zeigen sie, wie sogar die Sicherheitsgarantien untergraben werden können, die von speziellen Sicherheitsarchitekturen versprochen werden. Darüber hinaus werfen diese Angriffe ein Licht auf die besonderen Herausforderungen, die mit der Sicherung von Hardware/Mikroarchitekturen verbunden sind: Es ist wesentlicher (und schwieriger), die Hardware vor der Produktion umfassend auf Sicherheitsmängel zu untersuchen, da Hardware im Gegensatz zu Software nach der Herstellung nicht gepatcht/aktualisiert werden kann.

Hardware kann nicht mehr zuverlässig als Vertrauensbasis dienen, es sei denn, wir entwickeln und übernehmen neue Design-Paradigmen, bei denen die Sicherheit proaktiv angegangen und über die gesamten Schichten unserer Computerplattformen auf allen Hardware-Design- und Implementierungsebenen geprüft wird. Darüber hinaus müssen neuartige, flexible und sicherheitsbewusste Design-Mechanismen in die Mikroarchitektur von Prozessoren und in hardwaregestützte Sicherheitsarchitekturen integriert werden, die den Konflikt zwischen Leistung und Sicherheit praktisch lösen können, indem sie es ermöglichen, den Kompromiss an die gewünschten Anforderungen anzupassen.

In dieser Dissertation erforschen wir die Perspektiven und Auswirkungen an der Schnittstelle von Hardware und Sicherheit, die sich über die gesamten Schichten unserer Computerplattformen und System-on-Chips (SoCs) ergeben. Auf der einen Seite untersuchen wir, wie wir Hardware und ihre Vorteile im Vergleich zu Software nutzen können, um effizientere und effektivere Sicherheitserweiterungen zu entwickeln, die Sicherheitsarchitekturen dienen, z.B. um Attestierung und Integritätsschutz des Kontrollflusses anzubieten, mit dem Ziel, Software vor Angriffen zu schützen, die Software-Schwachstellen ausnutzen. Außerdem schlagen wir vor, dass die Sicherheitserweiterungen zur Laufzeit mikroarchitektonisch so konfiguriert werden, dass sie verschiedene Arten von Sicherheitsdiensten bereitstellen und sich so flexibel an unterschiedliche Einsatzanforderungen anpassen lassen. Außerdem untersuchen wir, wie wir diese hardwaregestützten Sicherheitsarchitekturen und -erweiterungen selbst vor mikroarchitek-

tonischen und Software-Angriffen schützen können, welche Designfehler ausnutzen, die ihren Ursprung in der Hardware haben, z.B. eine unsichere geteilte Nutzung von SoC-Ressourcen. Insbesondere konzentrieren wir uns in dieser Arbeit auf Cache-basierte Seitenkanalangriffe, für die wir elegante Cache-Designs vorschlagen, die diese Angriffe grundlegend entschärfen und gleichzeitig die Leistung erhalten, indem sie ermöglichen, den Kompromiss zwischen Leistung und Sicherheit zu konfigurieren. Wir untersuchen auch, wie diese Cache-Designs in flexible und anpassbare Sicherheitsarchitekturen integriert werden können, um diese zu ergänzen und so ein breites Spektrum neuer Anwendungen mit unterschiedlichen Leistungs-/Sicherheitsanforderungen zu unterstützen. Schließlich nehmen wir unsere Computerplattformen auch unterhalb der Design-Ebene unter die Lupe, indem wir untersuchen, wie die tatsächliche Implementierung dieser Mechanismen eine weitere potenzielle Angriffsfläche darstellt. Wir untersuchen, wie die Sicherheit von Hardware-Designs und -Implementierungen derzeit vor der Fertigung analysiert wird, und beleuchten gleichzeitig die grundlegenden Grenzen der modernen Hardware-Sicherheitsanalyseverfahren sowie das Potenzial für verbesserte und skalierbare Ansätze.

ACKNOWLEDGMENTS

A PhD dissertation is not possible without the support and guidance of advisors and mentors, and many fruitful collaborations with budding and experienced researchers. I want to thank many people, and probably won't be able to thank them all here by name.

First and foremost, I would like to thank my advisor Prof. Ahmad-Reza Sadeghi for his guidance and supervision during my PhD adventure. Ahmad gave me the opportunity to join the lab, to explore and research different angles in system security, collaborate with competitive researchers, and eventually pursue my research ideas in hardware-based security. He connected me with many of the prominent researchers in his network, which opened up many opportunities for me. He taught me the craft of competitive scientific research and publishing, and the art of presenting ideas coherently at different abstraction levels. Last but not least, he taught me mental resilience and perseverance, for which I will always be grateful.

I would also like to thank Prof. Dr. Farinaz Koushanfar for getting me started with the opportunity to work on the project that led to my very first publications when I joined the lab. Ever since, she has always supported me whenever I seek it. I have also been very fortunate to collaborate with Prof. N. Asokan, who has been a great mentor to me over the years. I look up to him, and try to always pay it forward at any mentoring chance I get. I would also like to thank Prof. Dr.-Ing. Lucas Davi for patiently guiding me along early on when I first joined, and for helping me learn the art of high-calibre scientific writing. I'd also like to thank Prof. Jeyavijayan Rajendran (JV) for collaborating and nurturing together the exciting Hack@DAC over the years.

This dissertation is based on results from collaborations with many researchers with whom I've had the privilege to collaborate and learn with and from. Many sincere thanks to my colleagues at the System Security Lab at TU Darmstadt, for collaborating and surviving together through one paper deadline after the next. I will miss all our exciting brainstorms, coffees, academic gossip, lunches, and conference adventures we've shared over the years! Without working together and taking a chance on each other and our ideas, our results wouldn't be possible. I have had the pleasure to collaborate with many external researchers and PhD candidates as well, whom I want to thank for all their hard and patient work that enabled our results and publications together.

Through our lab's collaborations with Intel and Intel Labs, I was fortunate to get to know and work with many people whom I want to thank: Matthias Schunter, Patrick Koeberl, Jason Fung, Arun Kanuparthi, Hareesh Khattri, and Anand Rajan.

Last, and definitely not least, this dissertation, and everything that led to it, wouldn't be possible without the patient love and support of my family, my husband, and my friends. Many sacrifices and compromises are necessary for a PhD journey to come to a successful conclusion, which doesn't work without the patient understanding of your dear ones. I want to sincerely thank them for always having my back through it all and bearing with me!

During my work on this thesis, I had the opportunity and honor to collaborate with exceptional researchers and PhD students at both the Technical University of Darmstadt and worldwide. I thank all my co-authors and collaborators for their contributions and our fruitful discussions that led to the publications that this thesis is based on.

Chapter 2

Chapter 2 is based on [37] (Appendix A), [144] (Appendix B), [38] (Appendix C), [105] (Appendix D), and [40] (Appendix E).

In [37] (Appendix A) **LO-FAT: Low-Overhead Control Flow ATtestation in Hardware [DAC'17]**, I led the work and focused on the implementation modules that capture and track the execution metadata from the processor pipeline. I contributed with co-authors Shaza Zeitouni and Thomas Nyman to the design discussions and security analysis that led to this publication. Shaza Zeitouni focused on the implementation modules that process the execution metadata and control the computation of the final attestation report. Andrew Paverd contributed to the discussions on the security of the scheme and Patrick Koeberl contributed to the discussions on the hardware architecture.

In [144] (Appendix B) **ATRIUM: Runtime Attestation Resilient Under Memory Attacks [ICCAD'17]**, I focused on the implementation that interfaces and integrates the scheme with the processor pipeline. I contributed with co-author Shaza Zeitouni to the design discussions and implementation that led to this publication. Shaza led this work and focused on the implementation of the proposed scheme. Orlando Arias and Dean Sullivan contributed to the attacks on state-of-the-art attestation schemes (SMART and C-FLAT). Ahmad Ibrahim contributed to the discussions on the security guarantees of the scheme.

In [38] (Appendix C) **LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution [ICCAD'18]**, I conceived the main idea and design, and led the project and implementation. Tigist Abera contributed to the static and dynamic binary analysis implementation for the verifier and the security analysis. Ahmad Ibrahim contributed to the discussions on the security analysis of the scheme.

In [105] (Appendix D) **HardScope: Hardening Embedded Systems Against Data-Oriented Attacks [DAC'19]**, I contributed with co-authors Thomas Nyman, Shaza Zeitouni and Aaro Lehikoinen to the discussions on the design and implementation that led to this publication. I focused on the design and integration of the HardScope Instruction Set Extension and the RSE HardScope hardware extension into the processor.

Thomas Nyman led the work and conceived the idea of the Run-time Scope Enforcement (RSE) and the design of the RSE Instruction Set Extension. Aaro Lehikoinen adapted the RSE Instruction Set Extension to RISC-V and implemented the support for the new instruction in GCC. Shaza Zeitouni focused on the design and implementation of the HardScope Instruction Set Extension hardware and evaluated the overheads of the hardware extension. Kesara Gamlath and Rangana De-Silva, under Thomas's supervision, ported the implementation on an FPGA and evaluated its performance. Thomas Nyman implemented the platform software support for HardScope to the processor software stack and evaluated its security.

In [37] **LO-FAT: Low-Overhead Control Flow ATtestation in Hardware [DAC'17]**, [144] **ATRIUM: Runtime Attestation Resilient Under Memory Attacks [ICCAD'17]**, and [105] **HardScope: Hardening Embedded Systems Against Data-Oriented Attacks [DAC'19]**, I focus, for my thesis, on the capabilities driven from leveraging existing processor (micro-)/architectural features, extensions and trusted hardware assumptions to enable more efficient protection for software with stronger security guarantees than software-based solutions. Co-author Shaza Zetouni focuses, for her thesis, on how relying on hardware trust anchors is used to establish and verify platform run-time integrity by providing different security services under different adversarial assumptions for embedded devices.

In [40] (Appendix E) **CHASE: A Configurable Hardware-Assisted Security Extension for Real-Time Systems, [ICCAD'19]**, I co-led the work with co-author Shaza Zeitouni and contributed to the discussions on the design and implementation that resulted in this publication. I focused on enabling a consolidated security extension that can be configured flexibly to adapt to different security requirements and deployment settings. In my thesis, I focus on leveraging trusted processor and complementary hardware to provide configurable software protection at run-time that can adapt to different application requirements. Shaza Zeitouni focused on enabling the attestation mechanism for securing timing-critical applications, and in her thesis, focuses on relying on the hardware trust anchors to enable run-time protection that can meet the requirements of timing-critical applications on embedded devices. Ahmad Ibrahim contributed to the discussions on the security guarantees and analysis of the scheme.

Chapter 3

Chapter 3 is based on [41] (Appendix F), [44] (Appendix G) and [11] (Appendix H).

In [41] (Appendix F) **HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments, [USENIX Security'20]**, I conceived the main idea and led the work, and focused on the hardware design, implementation and evaluation. I contributed with Tommaso Frassetto to the discussions on the design, implementation and security analysis that led to this publication. Tommaso focused on the implementation on the architectural simulator, and its performance evaluation starting from a

preliminary work prepared by a Bachelor student whom I supervised.

In [44] (Appendix G) **Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures, [To appear in NDSS'22]**, I conceived the main ideas and led the work. I focused on the architecture of Chunked-Cache, its hardware design, implementation and evaluation, and led the evaluation based on the architectural simulator. I contributed with co-author Emmanuel Stapf to the discussions on the Chunked-Cache design and security analysis that resulted in this publication. Emmanuel Stapf focused on the design and implementation of Chunked-Cache on the architectural simulator, its integration, and evaluation. Pouya Mahmoody focused on the architectural simulator setup and configuration for integrating and evaluating Chunked-Cache. Alexander Gruler contributed to the benchmarking on the architectural simulator.

In [11] (Appendix H) **CURE: A Security Architecture with CUstomizable and Resilient Enclaves, [USENIX Security'21]**, I contributed with co-authors Patrick Jauernig and Emmanuel Stapf to the discussions on the design and implementation that resulted in this publication. I focused on the design, implementation, and evaluation of the cache partitioning for CURE. Emmanuel Stapf conceived the main idea and led the work, and supervised the M.Sc. thesis of co-author Matthias Klimmek whose work focused on the software stack implementation of the CURE architecture and its evaluation. He 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. 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 paper writing.

Chapter 4

Chapter 4 is based on [39] (Appendix I).

In [39] (Appendix I) **HardFails: Insights into Software-Exploitable Hardware Bugs, [USENIX Security'19]**, all co-authors contributed to the work and discussions that led to this publication. I led the paper writing, and contributed to the Hack@DAC 2018 competition whose results this publication is based on, and to the technical insights and analysis required for the publication. David Gens contributed to the paper writing and the discussions on the paper contributions and structure. Patrick Haney contributed to the implementation required for the Hack@DAC 2018 competition. Garrett Persyn contributed to the hardware vulnerabilities detection analysis using formal verification techniques. Arun Kanuparthi and Hareesh Khattri contributed to the hardware security discussions and the design of the hardware vulnerabilities used in the competition.

CONTENTS

1 INTRODUCTION

For decades, *software security* has been the primary focus and concern when it comes to securing our computing platforms. Operating system and software vendors as well as academia have been exerting extensive efforts in hardening computing platforms against software-based attacks in the ever-evolving attacks-defenses arms race. Such attacks are usually exploiting various software and architectural security vulnerabilities, and are more commonly runtime attacks that exploit memory corruption vulnerabilities, e.g., buffer overflows. On the other hand, far less open scrutiny and efforts have been invested in *hardware and microarchitectural security*, despite their significant role both in inherently constituting the foundation of our computing platforms as well as in emerging hardware-enforced dedicated security mechanisms.

The long-established working assumption has been that the underlying hardware, provisioned by hardware and chip vendors, is trusted and secure. Only in recent years, did the recent uncovering of new types of security threats and attacks trigger a paradigm shift which disrupted the traditional threat assumptions which have, for long, considered software-only vulnerabilities and unjustifiably assumed the underlying hardware and our processors to be trusted. This emerging class of attacks are largely microarchitectural, and are usually *cross-layer* in nature, i.e., they usually involve unprivileged software remotely exploiting *hardware* vulnerabilities (design or implementation flaws) at different abstraction layers of the computing system to bypass existing protection mechanisms, thus achieving privileged code execution and accessing otherwise sensitive information. In recent years, such attacks have been shown to affect a wide range of computing platforms, ranging from low-end devices to high-end server systems of different architectures and vendors, e.g., Intel, AMD and ARM [83, 2, 90, 84, 81, 95, 143, 54, 48, 47, 88, 6, 5, 129, 101, 132, 134, 25, 27, 118, 20, 80, 129, 120, 90, 132, 54, 48].

The Root Causes. The root causes of these cross-layer attacks originate at different hardware abstraction layers such as design flaws at the microarchitectural layer [90, 83, 134, 25, 73, 118, 20, 107, 71, 75, 92, 140, 60, 142], and/or implementation flaws at the low-level hardware implementation layers [80, 129, 77, 120]. Typically, these attacks work by causing an otherwise concealed or temporary state of the hardware to become illegitimately visible to or compromised by unauthorized software, as we describe next.

Abstraction Layers of a Computing Platform. Figure 1 shows a simplified representation of the different abstraction layers of a computing platform. The *architecture layer* of a platform is a high-level abstraction layer which specifies the control unit and datapath (usually comprising the execution and logic units, the system memories and the general-purpose and architectural registers). The architecture layer also describes the instruction set of the platform and its expected behavior and representation to the *software*

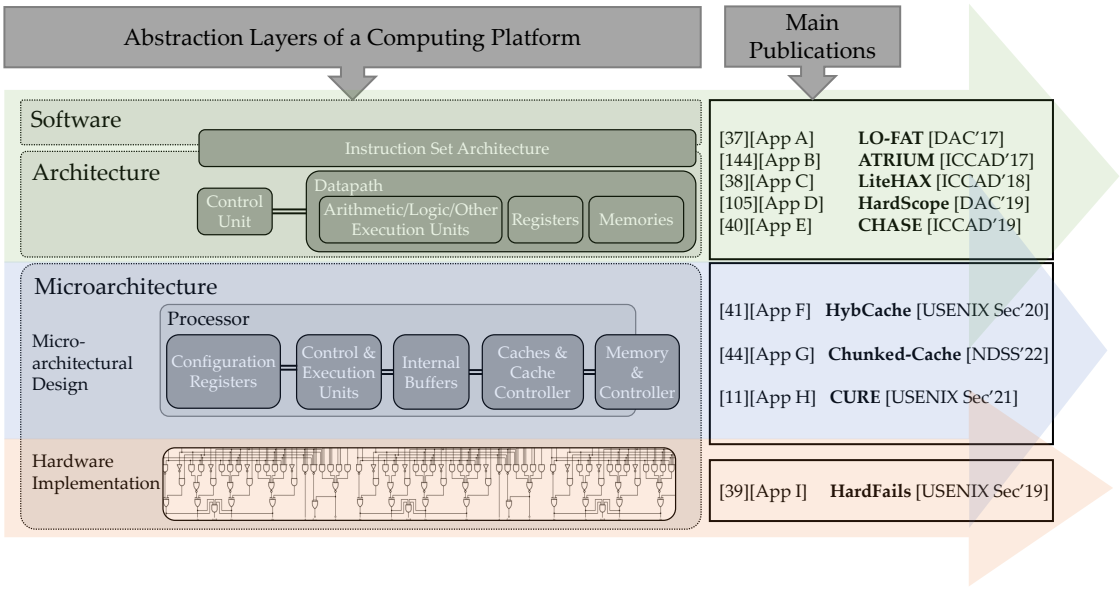


Figure 1: The different layers of a typical computing stack and a categorization of the main publications of this thesis across it

layer, i.e., the CPU architectural state and the contents of the general-purpose registers that should be visible to the software layer. The *microarchitecture layer* is a lower-level realization of that architecture and includes the design techniques leveraged to actually realize the processor architecture, e.g., the specific-purpose configuration registers, execution units, buffers, cache hierarchy and controller, and the ever-evolving performance optimizations of our modern processors. These architectural and microarchitectural designs and functionalities are then realized and implemented with concrete hardware logic gates, constituting the *hardware implementation layer*. While the architecture represents the current software-visible state, the microarchitectural state (and that of layers underneath) is only intermediate, can be out of sync, and is never intended to be visible to the software.

Cross-Layer Microarchitectural Attacks. At the microarchitecture layer, hardware vendors have been continuously enabling processors with closed-source and proprietary performance boosting optimizations, e.g., cache hierarchies and speculative execution. Speculative execution is an optimization technique where an out-of-order processor "speculates" that a branch or a load would use a value before the processor completes computing that this speculative value is correct, and execution proceeds with subsequent instructions "temporarily" or "transiently" assuming this speculated value. Once that branch/load resolves, the processor knows whether the speculative value was indeed correct. If correct, then executed instructions that depend on that value were validly executed and are retired, i.e., their results become architecturally visible to the software. Otherwise, the executed instructions that depend on that speculative value are invalid and are squashed, i.e., their results are not committed to the architectural pro-

gram state and almost most changes are reverted. While the visible architectural state is guaranteed to be unaffected, microarchitectural traces and "side effects"' are left behind that affect the microarchitectural state of the processor, such as secret-dependent cache lines fetched during this transient execution, which were not reverted or cleared up. The current cache state, after this transient execution, can be leaked (and made visible to the software layer) by an adversary by means of cache-based side-channel attacks such as Flush+Reload [60, 142] and Prime+Probe [107, 71, 75, 92, 140]. The adversary measures latencies of cache accesses and exploits the inherent latency discrepancy between hits and misses (timing side channel) to infer secrets that are otherwise inaccessible. This is only one simplified example of how microarchitectural security vulnerabilities can be exploited to launch cross-layer attacks. In recent years, such attacks have shaken our trust assumptions in the underlying hardware of our computing platforms and have been shown to have a devastating impact on their security assurances. More critically, they have been even shown to circumvent the security promises of dedicated hardware-assisted security architectures, e.g., Trusted Execution Environment (TEE) architectures [18, 119, 100, 52, 89, 145].

Even beneath the microarchitecture layer, at the hardware implementation layer, implementation flaws can also reside in the concrete hardware realization of any of the architectural or microarchitectural components and can have severe security implications. For example, the values of security-relevant configuration registers can be incorrectly read from the processor, a side-channel protected cache architecture implementation can be flawed, or access control for a debug interface can be incorrectly implemented in hardware.

The Challenges with Hardware and Microarchitectural Security. As processors and System-on-Chips (SoCs) scale in computing capability and complexity to keep up with the increasing computation and market demands, so do the underlying hardware and built-in optimizations. This further aggravates the challenge of discovering such cross-layer exploits and identifying their root causes where they actually originate, both at design-time and post-production. Moreover, these performance-boosting optimizations are usually proprietary and closed-source with no open documentation or accessibility to their inner workings, which has always been the case with hardware intellectual property (IP), except perhaps very recently with the growing advent of open hardware and RISC-V processors. However, the status quo remains largely unchanged for the biggest players in the semiconductor and processor industry and their proprietary platforms. Only through extensive reverse engineering efforts are researchers able to disclose how these microarchitectural implementations actually work and the security vulnerabilities therein.

Once these vulnerabilities and exploits are uncovered, contrary to software, patching the hardware post-production is not possible. The only feasible mitigation is to attempt to tackle these vulnerabilities with software and microcode "hotfixes" and patches. Microcode patching is limited to only a number of changes possible to the instruction set architecture, e.g., modifying the interface of individual complex instructions and adding or removing instructions [62]. Nevertheless, this always comes at the cost of a

performance regression, while for some vulnerabilities it is not even possible. For instance, microcode patching cannot mitigate the Spoiler attack [73]; this requires fixing the hardware of the memory subsystem at the hardware design phase, which is not feasible for legacy systems. In short, software and microcode patching may circumvent some of the resulting problems and provide "symptomatic" fixes. However, they do not fundamentally patch the flaw in the hardware where it originates.

Hardware and microarchitectural fixes and re-designs, on the other hand, promise a significantly smaller performance impact, while addressing the flaws fundamentally, but these require hardware modifications, which are only feasible for upcoming processor generations. Moreover, most proposed fixes are usually static and inflexible, i.e., they mitigate a specific issue with a specific microarchitectural design or approach that cannot adapt to mitigate emerging attacks, and cannot be configured to adapt to different adversarial settings or customized application-dependent performance/security requirements.

Thus, it follows intuitively that conducting an exhaustive security verification and analysis of hardware at *design-time* before production is even more critically required for hardware than software. Despite of this, state-of-the-art hardware security analysis techniques and methodologies severely lag behind the far more established spectrum of software security analysis techniques [78, 106]. Recently, however, inspired by software practices, the semiconductor industry has adopted a security development lifecycle (SDL) for hardware [63, 126]. This process comprises different techniques and tools, such as RTL manual code reviews, assertion-based testing, directed random testing combined with regression testing, dynamic simulation, and formal verification techniques. Nevertheless, the growing complexity of processor designs and the outbreak of cross-layer attacks described above represent difficult challenges for these security verification techniques, where they have been shown to fall short. Recent sophisticated attacks exploit complex and subtle inter-dependencies between the hardware and software. Thus, this requires verification techniques capable of modeling, capturing and verifying these different interactions accurately. Currently, state-of-the-art techniques lack in this respect. Moreover, they do not scale flexibly and in an automated manner with the growing size and complexity of real-world SoC design, thus still requiring extensive manual intervention and human expertise. In fact, the most significant challenge with hardware security analysis, as it stands now, is the prerequisite to anticipate potential security issues and requirements at design-time under the assumed threat model and to prepare and describe the relevant specifications. Assessing and analyzing the effectiveness of existing hardware security analysis techniques in detecting different classes of hardware vulnerabilities is another research question we investigate in this thesis and through the hardware security competitions we have organized.

1.1 OUR INSIGHTS

In light of these open challenges and problems with respect to the security of hardware, we summarize next our insights that inspire the vision of this thesis (described in Section 1.2) and propel its contributions.

- The underlying hardware of our platforms, being the foundation of our computing systems, must always constitute inherently all or a portion of the trusted computing base (TCB) of our computing systems.
- Furthermore, the hardware is often even leveraged in dedicated hardware-assisted security mechanisms and architectures to form the root of trust/trust anchor, e.g., in Intel SGX [68, 32], ARM TrustZone [7], Intel Control-Flow Enforcement Technology (Intel CET) [69], and Dover Microsystems' CoreGuard technology [99]. Leveraging the hardware in dedicated security mechanisms enables providing significantly more efficient security services when compared to their software counterparts, in terms of performance, as well as security guarantees under even stronger adversary assumptions, e.g., untrusted OS kernel and/or hypervisor in TEE architectures.
- Given the crucial role hardware plays, both fundamentally and in dedicated hardware-assisted security mechanisms, recent microarchitectural attacks that exploit hardware flaws break the trust assumptions in hardware, and consequently the derived security guarantees. Hardware, in this current state, cannot serve reliably as the root of trust in our computing systems anymore. This pressingly calls for the need to systematically rethink our hardware design paradigm, and exhaustively analyze and verify the security of the underlying hardware/microarchitectural, both at design and implementation, to restore justified trust in the hardware.
- These recent microarchitectural cross-layer attacks usually exploit flaws originating in performance boosting and other microarchitectural optimization features and interfaces in our modern processors. It is not practical, however, to forsake them altogether for the sake of security, where performance remains the ultimate market requirement. On the other hand, it is also not responsible, at least for a spectrum of use cases, to discard the resulting security concerns and implications with a sole focus on performance. It becomes necessary to investigate and develop new design paradigms and approaches that address this persistent conflict and enable configurable and flexible performance-security trade-offs *(micro-) architecturally* by design in our computing platforms. These would enable flexible and on-demand configuration or "tuning" of the level of security guarantees required along with the relevant performance cost this entails for the application in question, besides adapting to varying adversarial settings.
- Even beyond security flaws at the microarchitectural design, another security threat arises from the underlying implementation itself of these components. For example, while a partitioned cache is required microarchitecturally to mitigate resulting side-channel attacks, how this partitioning and cache management is actually implemented in hardware logic may still have security implications or generate other potentially exploitable side channels.
- As demonstrated by recent attacks, it is also becoming increasingly difficult for designers to keep up with the growing complexity of hardware. State-of-the-art

hardware security analysis techniques are fundamentally challenged in efficiently and effectively uncovering different types of hardware vulnerabilities at design time. New techniques are required to uncover more hardware vulnerabilities and side-channel leakages before production, especially since hardware flaws cannot be patched after production and deployment.

1.2 THESIS VISION AND SCOPE

The problem of hardware/microarchitectural security and its multifaceted challenges and implications, as outlined above, imply that hardware, in its current state, cannot reliably serve as the root of trust of our computing platforms anymore. This calls for fundamentally rethinking the design and security analysis of our computing platforms altogether to tackle today's and future security challenges.

1.2.1 *Vision*

We envision the pressing need for new design paradigms where security is proactively addressed and scrutinized across the full stack of our computing platforms, at all hardware design and implementation layers. Furthermore, the security of the interactions of the hardware with the overlying software and all relevant subtleties should also be scrutinized and analyzed by scalable and efficient means. We propose that flexible security-aware design mechanisms are incorporated within the different processor microarchitectural units and optimization features, as well as hardware-assisted security mechanisms and architectures, where these mechanisms can be configure the pertinent security/performance guarantees as desired. This would enable computing platforms to adapt flexibly to different applications and practically address the inherent conflict between performance and security by allowing that the trade-off is calibrated by means of efficient design mechanisms. Moreover, more rigorous and efficient full-stack cross-layer information flow/security analysis techniques are required to complement the security-aware design paradigms, in order to vet the hardware design and implementation as well as its overlying software for security vulnerabilities and design flaws.

In short, the security of the underlying hardware cannot be tackled as an after-thought anymore. Designing, implementing and testing the hardware with security in mind, especially in dedicated security architectures, must become integral to the hardware development lifecycle.

1.2.2 *Scope*

Towards realizing this vision, we take first steps in this thesis and focus primarily on exploring the implications and potential roles of the underlying hardware and microarchitecture on systems security. On one front, we 1) investigate and propose a suite of different hardware-based security mechanisms and extensions that can serve security architectures by providing different software security services to mitigate different classes

of software attacks. We show how they are significantly more efficient, perform better and provide stronger security guarantees than their software counterparts. We also investigate how they can be microarchitecturally configured at runtime to provide different services, e.g., control-flow integrity or runtime execution measurement. The thesis contributes to this front with the following five publications that can be found in Appendices A, B, C, D, and E:

> [37] 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 IEEE/ACM Design Automation Conference (DAC). ACM, 2017. Core Rank A. Appendix A.
>
> [144] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. **ATRIUM: Runtime Attestation Resilient under Memory Attacks.** In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2017. Core Rank A. Appendix B.
>
> [38] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. **LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution**. In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2018. Core Rank A. Appendix C.
>
> [105] 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 IEEE/ACM Design Automation Conference (DAC). ACM/IEEE, 2019. Core Rank A. Appendix D.
>
> [40] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, and Ahmad-Reza Sadeghi. **CHASE: A Configurable Hardware-Assisted Security Extension for Real-Time Systems**. In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2019. Core Rank A. Appendix E.

On a second front, to protect the underlying hardware of these security extensions and our computing platforms from microarchitectural attacks, we 2) analyze how microarchitectural design flaws that originate in the hardware, e.g., timing side channels, can be mitigated by redesigning the hardware fundamentally with flexible security/performance configurable mechanisms built in. We also show how these can be integrated into flexible and customizable security architectures, thus enabling their customization even further to support a wide spectrum of different and emerging applications with different performance/security requirements. The thesis contributes to this front with the following three publications that can be found in Appendices F, G, and H:

> [41] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. **HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments**. In USENIX Security. USENIX Association, 2020. Core Rank A*. Appendix F.

[44] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures**. In Annual Network and Distributed System Security Symposium (NDSS), 2022. Core Rank A*. Appendix G.

[11] 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. USENIX Association, 2021. Core Rank A*. Appendix H.

Lastly, we investigate and show 3) how the actual implementation of such hardware-assisted security extensions and mechanisms as well as processor and SoC microarchitecture is also a potential attack surface. We investigate how hardware designs are currently analyzed for their security and provide potential research directions on how this can be improved. The thesis contributes to this front with the following publication that can be found in Appendix I:

[39] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi and Jeyavijayan Rajendran. **HardFails: Insights into Software-Exploitable Hardware Bugs**. In USENIX Security. USENIX Association, 2019. Core Rank A*. Appendix I.

1.3 MAIN CONTRIBUTIONS

More specifically, we categorize our contributions as described above along the different layers of a typical computing stack into three main pillars as shown in Figure 1, namely:

1. Hardware-based security mechanisms
2. Secure microarchitecture design for trusted execution
3. Hardware implementation security

We illustrate the categorization of these contributions into the thesis chapters and the relevant publications in Figure 2, and present a more detailed overview of each next.

Hardware-based security mechanisms. Conventional remote attestation allows a trusted party to establish trust in a potentially compromised and untrusted embedded device by statically verifying that the program binary initially loaded is unmodified. However, it cannot provide any guarantees with respect to the *execution behavior* of this program, e.g., it cannot detect any runtime attacks that hijack the control or data flow of execution. These runtime attacks conventionally exploit a security vulnerability, typically memory corruption, and modify the code on a device by injecting malicious code. However, protection mechanisms such as Data Execution Prevention (DEP) [61] have proven effective against code-injection attacks, thus attackers have resorted to other attack techniques which rely on code reuse, such as Return-Oriented Programming (ROP) [121] and Jump-Oriented Programming (JOP) [14]. These techniques exploit vulnerabilities to corrupt control data and re-use the code chunks already residing in the

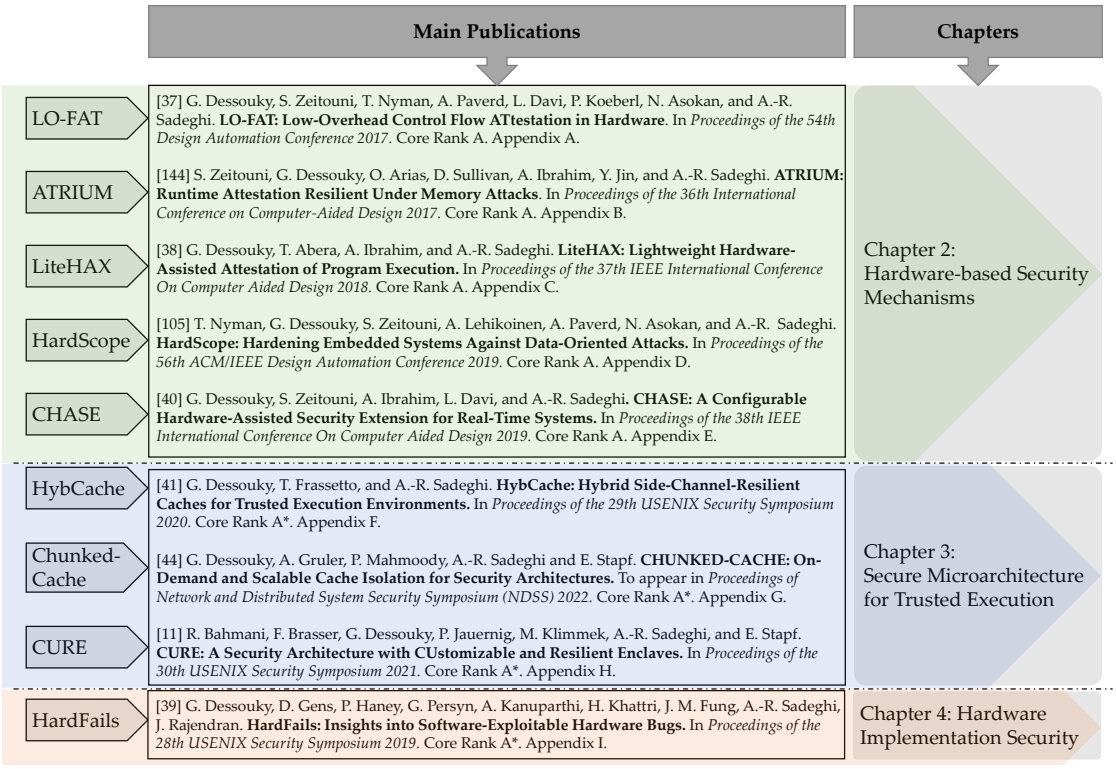


Figure 2: Main publications of this thesis and the thesis structure

memory of the vulnerable program to build the attack payload and hijack the control flow of the program.

Hence, a software-based *runtime* remote attestation mechanism was first proposed that can capture and detect such control-flow attacks [4]. However, being software-based in design and implementation, it requires instrumentation of the source code and incurs prohibitively high performance overheads that are largely application-dependent. We investigate and show through our work how these limitations of software-based security mechanisms can be overcome by relying on hardware instead. We present LO-FAT [37] (Appendix A), the first hardware-based mechanism for control-flow attestation. LO-FAT works by leveraging existing processor hardware features that inherently keep track of execution in a cycle-accurate manner to track execution and compute a hash measurement over the it. The computed values are communicated securely to a trusted third party to verify the control flow of the execution. This hardware-based approach enables significantly more efficient control-flow attestation in contrast with the software-based scheme, besides providing stronger security guarantees while relying on a significantly smaller TCB than the software-based counterpart.

The security guarantees of LO-FAT (as well as conventional static remote attestation schemes) rely on the assumption that attacks are software-only and that the program code cannot be modified at runtime. In our work [144] (Appendix B), we demonstrate how these assumptions may, in practice, not hold where a stronger adversary is capable of modifying the code memory such that benign code is attested but malicious code is

executed. This would effectively bypass the attestation mechanism in place and leave the device vulnerable to Time of Check Time of Use (TOCTOU) attacks. To mitigate these attacks, we present a hardware-based security extension, called ATRIUM, that provides a variant of runtime attestation that securely attests both the code's binary and its execution behavior at a finer granularity. In doing so, it can effectively mitigate these memory manipulation attacks described above.

Both of these schemes, however, as well as C-FLAT remain vulnerable to the more sophisticated data-oriented programming (DOP) attacks [64]. Such attacks subvert these defense schemes by keeping the control flow and the binary of the code unmodified, while still enabling Turing-complete malicious execution by carefully corrupting only non-control data to stitch a sequence of operations on attacker-controlled input. Prominent defenses, e.g., control-flow integrity (CFI) [3, 69], code-pointer integrity [85], and (fine-grained) code randomization [86] to name some, fall short in mitigating these sophisticated attacks. In our next work [38] (Appendix C), we investigate how to provide an efficient hardware-based remote attestation mechanism for RISC-based devices, called LiteHAX, that can additionally detect non-control-data attacks. LiteHAX allows to capture and record both the control- and data-flow events of a program executing on a remote device and report them to a trusted verifying party. LiteHAX works, in principle, by interfacing with the processor pipeline and capturing the execution of memory access instructions at runtime directly from the processor in parallel to the actual execution.

While runtime attestation of memory access operations, as shown in LiteHAX, is one hardware-based mitigation approach to detect DOP attacks after their occurrence, runtime *enforcement* of certain policies or constraints is another promising approach that blocks the attacks before they even occur. In another work [105] (Appendix D), we propose runtime scope enforcement to efficiently mitigate all currently known DOP attacks by identifying the lexical scope rules of variables at compile time and extracting memory safety constraints from them and enforcing them at runtime. To prototype our scheme, we presented HardScope [105] (Appendix D), a hardware-assisted runtime scope enforcement scheme for RISC-V based systems, that provides fine-grained context-specific memory isolation within programs.

The different hardware-based security mechanisms we have presented above, besides other state-of-the-art approaches proposed in academia and adopted in industry, either apply enforcement or execution tracking/attestation. Moreover, each assume different adversarial capabilities, thus mitigating only specific classes of attacks. No consolidated defenses exist that can be *configured* flexibly within the platform at runtime to thwart different adversarial capabilities depending on the desired security/functionality requirements and deployment environment. This is particularly a challenge for these hardwired hardware-assisted security extensions which cannot be upgraded or updated after fabrication (in contrast to software). This makes it impractical for system architects to deploy these hardware-assisted mechanisms in embedded platforms, despite their advantages over software-based defenses. In our work [40] (Appendix E), we present and discuss these insights and challenges in more detail, and present a consolidated runtime-configurable security extension, called CHASE. CHASE can be more flexibly adapted to provide different security guarantees and services at runtime, e.g., either enforcement

or more detailed execution tracking and attestation, depending on the desired security guarantees and the system real-time, availability and functionality requirements. This enables the adoption of such hardware-based security extensions and their customization at runtime to calibrate the security vs. performance trade-off for individual use cases and deployment settings.

We present a more detailed overview of our contributions described above in Chapter 2.

Secure microarchitecture for trusted execution environments. As described earlier, modern multi-core processors are augmented with various performance optimization features that make them vulnerable to a wide spectrum of different microarchitectural attacks, as shown in recent works [83, 2, 90, 84, 81, 95, 143, 54, 48, 47, 88, 6, 5, 129, 101, 132, 134, 25, 27, 118, 20]. Shared cache resources are one of the most popular optimization features, and also the most exploited in these attacks. Inherent timing differences between a cache hit and a cache miss in shared cache behavior are exploited to infer information on the victim's execution patterns, ultimately leaking private information such as a secret key or other confidential information [56, 53, 57, 55, 18, 147, 92, 16, 71]. The root cause for these attacks is mutually distrusting processes sharing the cache entries and deterministic and consistent set-associative eviction and access of these entries. Recently proposed defenses in academia and industry can be classified as randomization-based [137, 94, 92, 130, 115, 116, 138, 128] or partitioning-based [50, 79, 145, 93, 33, 58, 148, 76, 139, 87, 11, 136, 82, 137]. Recently proposed defenses based on randomized mapping of addresses to cache lines have been shown vulnerable to newer attack algorithms [116, 17, 113, 114] besides relying on weak cryptographic primitives [15, 114], and are generally designed to mitigate only certain classes of attacks. When customizing them to mitigate more advanced attack algorithms, they impose prohibitively high performance overheads [116]. Ultimately, they fail to provide well-grounded security guarantees because they do not fundamentally address the root cause for these attacks, namely, mutually distrusting code sharing cache resources. On the other hand, most partitioning-based 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, does not scale to support a larger number of protection domains, degrades performance for larger workloads, and may cause cache underutilization [137, 136, 82, 58]. More importantly, all such defenses assume that side-channel-resilient cache behavior is required for the entire execution workload and do not allow the possibility to selectively and flexibly configure the mitigation only for the security-critical portion of the workload.

To address these limitations of existing cache designs and provide a configurable and flexible side-channel-resilient cache microarchitecture for security architectures, we propose a flexible and soft partitioning of set-associative caches by means of a hybrid cache architecture, called HybCache [41] (Appendix F). HybCache can be configured to selectively apply side-channel-resilient cache behavior only for isolated execution domains that require this sophisticated security guarantee, while providing the non-isolated execution with conventional cache behavior, capacity and performance. An isolation do-

main is defined as any form of compartmentalization of the workload, e.g., a Trusted Execution Environment enclave (e.g., as in SGX or TrustZone).

While HybCache enables configurable cache side-channel resilience while maintaining non-degraded performance for the non-isolated execution, it still does not fundamentally mitigate all side-channel leakage. The cache occupancy side channel, where the adversary can attempt to infer the working set size of the victim, is the only side-channel leakage that is not mitigated by the HybCache construction. This leakage is inherently available in any cache architecture where the attacker and the victim processes compete for entries in shared cache resources. It can only be effectively blocked by strict cache partitioning, which we deliberately do not provide in the HybCache construction. This allows different isolation domains to still compete for cache entries, thus preserving dynamic cache utilization for the entire workload and unaffected performance for non-isolated execution.

In a follow-up work [44] (Appendix G), we propose another cache design, Chunked-Cache, that blocks this cache occupancy leakage by providing strict cache partitioning thus providing clean isolation, while still maintaining flexible cache utilization. Chunked-Cache enables an execution context to "carve" out its exclusive cache chunk of configurable capacity only if it requires cache side-channel resilience. When side-channel resilience is not required, mainstream cache resources can be freely utilized. This addresses the security-performance trade-off by efficiently enabling on-demand cache side-channel resilience, i.e. only when actually required, while providing well-grounded future-proof security guarantees.

Our work in secure cache designs has enabled more flexible and configurable cache-based side-channel security that can be adapted on-demand for different portions of the execution workload individually and independently. To further extend this configurability and flexibility to trusted execution capabilities generally, we focus next on the encompassing security architecture itself. Security architectures providing Trusted Execution Environments (TEEs) aim to protect sensitive services by compartmentalizing them in isolated execution contexts, called enclaves. However, existing TEE solutions suffer from critical shortcomings with respect to both security and functionality. They adopt a rigid approach where only a single enclave type is available, although, in fact, more flexibility is required, since different services require different types of enclaves that can adapt to the demands of the service in question. Moreover, they cannot even efficiently support emerging applications, e.g., machine learning services, which require secure binding of specific enclaves with specific peripherals (e.g., accelerators), or the computational power of multiple cores securely. Finally, their protection mechanisms against side-channel attacks, e.g., cache side-channel attacks, are either an afterthought "hotfix" or impractical for flexible usage, e.g., fine-grained allocation of cache resources to individual enclaves is usually not supported by default.

We investigate these shortcomings and challenges in our work [11] (Appendix H), and propose CURE, the first security architecture, which addresses these challenges by providing different types of enclaves whose boundaries can be flexibly configured and resources can be selectively allocated to them. Supported enclaves in CURE can either provide isolation either vertically within any single execution privilege level (sub-

space enclave), or across multiple privilege levels (kernel-space enclaves) or only for unprivileged applications (user-space enclaves). In doing so, CURE already outperforms the state of the art (at time of writing) in TEEs which usually provide only one type of enclave, as stated earlier. CURE also allows that system resources, e.g., peripherals, CPU cores, or cache resources are exclusively and selectively assigned to single enclaves, thus providing the desirable fine-grained resource allocation as well as on-demand and flexible side-channel protection.

We present a more detailed overview of our contributions described above in Chapter 3.

Security of hardware design and implementation. In this work, we investigate the security of hardware implementation itself, one level beneath microarchitecture/design decisions and side-channel mitigation mechanisms. In other words, while a hardware-based security extension can aim to address software vulnerabilities or mitigate side-channel leakage, such as cache partitioning, it may in fact be *incorrectly* implemented, thus compromising its promised security guarantees. Such hardware-based extensions and mechanisms, similar to the ones we have designed and developed during the course of this PhD [37, 144, 40, 38, 105] (Appendices A, B, C, D, and E) as well as industry solutions such as SGX [68, 32] and TrustZone [7], are not designed to ensure security at the hardware implementation level. Unless their implementations are exhaustively verified to ensure that they adhere to the formally defined desired security properties, they remain vulnerable to potentially undetected hardware bugs committed at design-time. Such hardware vulnerabilities can occur due to: (a) incorrect or ambiguous or incorrectly described/formalized security specifications, (b) incorrect design, (c) flawed implementation of the design, or (d) a combination thereof. Hardware implementation bugs can be introduced either through human error or by faulty compilation/synthesis of the design to its gate-level equivalent.

Unlike software flaws, hardware vulnerabilities committed at design-time cannot be fundamentally patched once the hardware is manufactured. This makes hardware security testing for detecting these bugs at design-time even more critically crucial than software security testing. The semiconductor industry already leverages an extensive variety of techniques, such as simulation, emulation, and formal verification to detect such bugs. While a rich body of knowledge and expertise is long established for software security, security-focused hardware testing and analysis are currently still lagging behind [78, 106]. To catch up, the industry has recently adopted a security development lifecycle (SDL) for hardware [126]. This process combines different techniques and tools, such as RTL manual code audits, assertion-based testing, dynamic simulation, and automated security verification. In spite of this, our underlying hardware remains vulnerable as demonstrated by the recent outbreak of cross-layer attacks [83, 2, 90, 84, 81, 95, 143, 54, 48, 47, 88, 6, 5, 129, 101, 132, 134, 25, 27, 118, 20] where both hardware-only vulnerabilities as well as intricate and complex hardware-software interactions have been exploited to mount these attacks. Existing industry-standard techniques are fundamentally limited in modeling and verifying these subtle hardware-software interactions, and thus fail to detect such flaws. Moreover, they also

do not scale adequately with the ever-growing size and complexity of real-world SoC designs.

Thus, in this work [39] (Appendix I) we take a closer look into the design and security assurance lifecycle of hardware, and focus the spotlight on the limitations and challenges of state-of-the-art hardware security verification. The first step in qualitatively assessing the effectiveness of existing verification techniques was to construct the test harness itself, i.e., the System-on-Chip (SoC) design and the bugs therein. Together with our industry partners and collaborators at Intel, we systematically constructed a varied set of 31 hardware register transfer-level (RTL) bugs inspired from their first-hand experience with bugs that they have encountered themselves at Intel, as well as public Common Vulnerabilities and Exposures (CVEs) and real-world errata [103, 104, 102, 90, 83, 62]. We injected the bugs into two open-source real-world RISC-V-based SoC designs, Pulpino [111] and PULPissimo [112]. We organized the first edition of what is now the largest international hardware security competition, Hack@DAC, in 2018 where 54 teams of researchers competed for three months to detect these bugs in the SoCs. We analyzed the results, and the nature of the approaches they relied on to detect these bugs, and which classes of bugs were not successfully detected by the teams.

In a second in-house investigation, we focused on two state-of-the-art formal verification tools (Formal Property Verification (FPV) [24] and JasperGold's Security Path Verification (SPV) [23]) to assess their effectiveness in detecting these bugs and their ease of use and friendliness. These represent the state of the art in hardware security verification and are used widely by the semiconductor industry.

Both the results of the competition and our investigation with formal verification tools have revealed that certain properties of RTL bugs can make them significantly more challenging to detect, both by manual inspection as well as formal verification techniques. Building on our findings from both case studies, we attempt to systematically classify and identify these bugs that are more challenging to detect and the characteristics that they have in common, where we call such bugs HARDFAILS [39] (Appendix I). We present our insights and findings in more detail in our work [39] (Appendix I).

Ultimately, our work and insights manifest why further research is urgently required to improve state-of-the-art security verification and analysis of hardware, and sheds light on potentially promising directions, e.g., hybrid techniques that combine both formal verification and simulation-based testing that would scale better than formal verification only, as well as more efficient testing inputs generation techniques, such as fuzzing. We presents our insights for future promising directions in this domain in more detail in Chapter 5.

Ever since it was first launched in 2018, we have been organizing Hack@DAC every year, and organized its first USENIX Security sequel, Hack@Sec, in 2020. Over the past few years, the competition has been growing in sophistication, size and popularity among both academics and industry professionals. The focus of the competitions has also shifted and evolved from only bug detection and root cause analysis in 2018 and 2019 to more interestingly tooling, automation and proof-of-concept exploitation in

Hack@DAC and Hack@Sec 2020. We discuss our work and insights with the competition and how it has evolved over the years in more detail in Chapter 4.

1.4 OTHER CONTRIBUTIONS

During the course of this PhD, several other contributions in complementary directions have also emerged that are are not included as core of this thesis. We present a brief overview of them next.

Hardware synthesis for secure computation. Through our work, we have shown how leveraging hardware synthesis tools can enable more efficient secure computation with Yao's garbled circuits protocol [141] and the protocol of Goldreich-Micali-Wigderson (GMW) [51], which work by evaluating a Boolean circuit that represents the desired functionality. Many works have thus focused on the practical design and generation of correct Boolean circuits to enable more efficient circuit-based secure computation in different adversarial settings. However, the complexity and time required quickly escalate for larger and more complex applications, e.g., floating-point arithmetic and signal processing. Moreover, the functional correctness of these circuits becomes more difficult to verify, making them more error-prone. Besides compromising functionality, faulty circuits may also compromise the security of the underlying applications, e.g., by leaking information about a party's private inputs. Therefore, an automated mechanism for generating correct large-scale circuits which can be used by non-expert developers is desirable to enable the practical adoption of secure computation protocols. TinyGarble [124] adopted a radical approach to this open challenge by leveraging long-established powerful hardware logic synthesis tools and customizing them to be adapted to automatically generate Boolean circuits for functions to be evaluated by Yao's garbled circuits protocol.

In our work [35], we further advance the deployment of these tools for secure computation, and show how to automatically use them to synthesize an extensive set of size-optimized circuits of basic and complex operations for Yao's garbled circuits protocol, as well as depth-optimized circuits for the GMW protocol. We build libraries of these optimized sub-block circuits and use these to automatically construct more complicated functionalities in a modular fashion, which would otherwise be impossible to build and optimize by hand. To also enable the generation of Boolean circuits for more complex functionalities such as floating-point arithmetic, which would otherwise be impossible by hand, we also leverage built-in Intellectual Property (IP) libraries (which are already extensively tested and verified) in commercial hardware synthesis tools. We extensively evaluate and benchmark our circuit constructions and show how we outperform the state of the art at the time of writing.

Next in [125], we present and prototype GarbledCPU, the first configurable general-purpose sequential CPU for 2-party Garbled Circuits-based secure sequential function computation. GarbledCPU provides support for secure function evaluation (SFE) with different privacy settings to allow for a configurable trade-off between privacy and performance that can be adapted according to requirements. The parties can choose to evaluate a private, semi-private or public function by revealing none, partial or all informa-

tion about the function respectively while still exploiting the advantages and simplicity of programming a processor.

Current two-party secure computation protocol implementations against passive adversaries generate and process data much faster than it can be communicated over the network. In [36], we introduce, deploy and evaluate novel methods to further reduce the communication bottleneck and round complexity of semi-honest secure two-party computation. We first improve communication for Boolean circuits with 2-input gates by factor 1.9x when evaluated with the GMW protocol. Furthermore, we evolve the conventional Boolean circuit representation from 2-input gates to a more compact multi-input/multi-output lookup tables (LUTs), thus enabling the evaluation of more complex functions by representing them into LUT-based circuits. We construct and propose two protocols for evaluating LUT-based circuits which offer a trade-off between online communication and total communication. Our most efficient LUT-based protocol reduces the communication overhead and round complexity by a factor 2-4x for several basic and complex operations compared to prior work. Since we evolve the protocols to evaluate LUT-based circuits, we also required new optimized LUT-based circuit representations of pertinent functions. We develop an automated toolchain that transforms high-level function descriptions into their LUT representations, where we re-purpose hardware synthesis tools for secure computation. We focus on LUT-based synthesis tools (often targeting FPGA-based development) in this work, which we customize and adapt to automatically generate optimized multi-input multi-output LUT representations. We demonstrate the improved efficiency and practicality of our LUT protocols by extensively evaluating them over a wide range of functionalities.

Besides the impact of our work to enable more practical and efficient secure communication [35, 125, 36], it additionally serves as concrete testimony to how knowledge from one discipline, i.e., hardware design and synthesis, can prove radically useful and enhancing for another discipline, i.e., efficient secure computation protocols.

Publications

[35] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. *Automated Synthesis of Optimized Circuits for Secure Computation.* In ACM Conference on Computer and Communications Security 2015 (CCS'15).

[125] Ebrahim Songhori, Shaza Zeitouni, Ghada Dessouky, Thomas Schneider, Ahmad-Reza Sadeghi and Farinaz Koushanfar. *GarbledCPU: A MIPS Processor for Secure Computation in Hardware.* In the Annual Design Automation Conference 2016 (DAC'16).

[36] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. *Pushing the Communication Barrier in Secure Computation using Lookup Tables.* In the Annual Network and Distributed System Security Symposium 2017 (NDSS'17).

Collective remote attestation. While remotely attesting the software integrity of a single device is well established (as we also show through our work in hardware-based attestation), scaling this service to a network of devices poses a multitude of research and deployment challenges. In this work [34], we systematically analyze and design collective remote attestation schemes, with the goal to overcome the limitations of prior schemes that were designed in an ad-hoc reactive fashion. Ultimately, we aim to provide a systematic foundation for collective remote attestation schemes that serves as reference design guidelines for researchers and practitioners. In doing so, we explore the design space for collective remote attestation and formally define and model notions of the efficiency, soundness and security requirements according to a given application domain. In light of these requirements, we also present, prototype and evaluate a concrete collective remote attestation scheme and show that it adheres to the aforementioned desirable requirements.

Publication

> [34] Ivan De Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. *Towards Systematic Design of Collective Remote Attestation Protocols.* In the 39th International Conference on Distributed Computing Systems 2019 (ICDCS'19).

Security of multi-tenant FPGA computing. Given the increasing deployment of Field Programmable Gate Arrays (FPGAs) in data centers, and their continuously evolving size, complexity and capabilities, researchers have proposed that they can be shared *spatially* by multiple tenants, or clients, simultaneously. In contrast to temporal sharing where the FPGA instance is shared but utilized by only one tenant at any point in time, in spatial multi-tenant deployment the FPGA fabric is simultaneously shared among mutually distrusting tenants. This can be enabled by leveraging the partial reconfiguration capability of FPGAs.

In [43], we systematically analyze prior research work on multi-tenant FPGAs in cloud computing at time of writing. We outline their adversary and deployment assumptions, acclaimed security guarantees, and analyze how they fall short with respect to both security and privacy. We also focus more specifically on categorizing existing works that demonstrate a new class of remotely-exploitable physical attacks on multi-tenant FPGAs by malicious tenants when they are sharing physical FPGA resources with the victims. Through investigating end-to-end multi-tenant FPGA deployment comprehensively, we reveal how these remote attacks, in fact, represent only one dimension of the security/privacy problem. Various more fundamental security and privacy challenges remain open and unaddressed in deploying multi-tenant FPGAs in cloud computing settings, which we investigate in this work. We also provide our insights on the most vital research challenges and open opportunities in the future of secure FPGA-based cloud computing. In doing this, we draw analogies with conventional CPU-based computing and outline the lessons learned that can proactively serve and guide the establishment of secure multi-tenant FPGA-based computing in the cloud.

Publication

> [43] Ghada Dessouky, Ahmad-Reza Sadeghi, and Shaza Zeitouni. *SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities.* In IEEE European Symposium on Security and Privacy 2021 (EuroS&P'21).

Security architectures for Network-on-Chips (NoCs). Novel many-core chips, designed to cater for the increased performance and computational power demands of emerging applications, require Network-on-Chip (NoC) based architectures to enable scalable and efficient communication among this increasing number of cores. However, NoC designs lack adequate security mechanisms that scale to provide the required security guarantees, such as enforced isolation of execution and resources, while preserving the desired scalable and efficient communication. New security-aware architectures that protect sensitive services in isolated or trusted execution environments, i.e., enclaves, usually target only multi-core designs, and thus cannot directly extend and scale to support NoC platforms. These architectures usually lack secure and flexible enclave-device binding and do not provide flexible and practical enclave memory management.

We investigate and address these fundamental challenges in our work [42], where we introduce and evaluate a new hardware security primitive, the Distributed Memory Guard, which performs NoC-level access control on outgoing memory requests. In doing so, we also investigate the memory fragmentation that occurs for typical cloud services, and realize that it is unavoidable on long-running systems. We analyze this specifically for enclave computing and highlight why this is one of the most significant challenges of enclave architectures that has not been practically addressed. Furthermore, this becomes a more critical challenge when scaling enclave computing to heterogeneous NoC-based architectures, thus motivating our work.

Publication

> [42] 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 the Annual Design Automation Conference 2021 (DAC'21).

1.5 THESIS OUTLINE

This thesis is structured in line with Figure 1, and consists of four subsequent chapters, where the first three chapters present our main contributions as shown in Figure 2. We conclude our work in the last chapter with a summary and on outlook for future research directions.

1. Hardware-based Security Mechanisms (Chapter 2)
2. Secure Microarchitecture for Trusted Execution (Chapter 3)
3. Hardware Implementation Security (Chapter 4)

4. Conclusion (Chapter 5)

Chapters 2, 3, and 4 each briefly introduces the problem statement that our work addresses, and presents a more detailed overview of our contributions, followed by the relevant peer-reviewed publications.

2

HARDWARE-BASED SECURITY MECHANISMS

2.1 PROBLEM STATEMENT AND MOTIVATION

The proliferating trend of the Internet of Things (IoT) and their increasingly collaborative nature has made all classes of computing systems, especially embedded devices, increasingly ubiquitous in a variety of different settings. While these devices collect, process, and communicate plenty of security/privacy/security-critical data, their pervasiveness, connectivity, and sensing/actuating capabilities render them increasingly vulnerable to a large spectrum of different attacks. On the other hand, to adhere to the desired cost and power consumption budgets as well as performance and deployment requirements, embedded devices are usually resource-constrained and lack the sophisticated security mechanisms often deployed in higher-end computing devices. This has made it particularly challenge to harden embedded devices security in the face of various known and emerging attacks, e.g., malware infestation, as well as runtime attacks such as control-flow hijacking [121, 14] and data-oriented programming (DOP) attacks [64]. Critical attacks exploiting embedded devices have been shown over the last decade, e.g., Stuxnet [31] and Mirai [67]. Such attacks commonly attempt to exploit software memory corruption vulnerabilities, e.g., buffer overflow vulnerabilities, to compromise the device. We briefly introduce these attacks next in subsection 2.1.1 and refer the reader to our work [37, 144, 38, 105, 40] (Appendices A, B, C, D, and E) for a more detailed description of the attacks.

2.1.1 *Software Runtime Attacks*

Traditionally runtime attacks exploit a security vulnerability, typically a memory corruption vulnerability, in order to modify the program code on a device by injecting malicious code. However, with the advent of W⊕X memory access policies such as Data Execution Prevention [61], code-injection attacks have been effectively mitigated. Thus, attackers have had to resort to other stealthier and more sophisticated tactics such as code-reuse techniques [127], e.g., Return-Oriented Programming (ROP) [121] or Jump-Oriented Programming (JOP) [14]. These techniques exploit memory vulnerabilities in order to corrupt control-data and be able to re-use code chunks or gadgets already residing in the memory of the targeted program and hijack the control flow of the program to construct the attack payload. In other words, the code binary is unchanged, but it is how the code gadgets are executed and their sequence that is actually compromised.

A stealthier class of attacks is that of data-oriented attacks [28], where a non-control-data variable is compromised to divert control flow to yet another valid execution path, but an illegal or unauthorized one in this particular execution context. More recently, even more sophisticated Data-Oriented Programming (DOP) attacks [64] were shown

which allow the adversary to execute Turing-complete malicious execution by corrupting only non-control-data to stitch a sequences of operations on attacker-controlled input. DOP attacks neither modify the binaries nor divert the program's control flow, thus rendering them significantly more challenging to mitigate or detect.

2.1.2 *Limitations of Existing Defenses*

To mitigate these sophisticated attacks, extensive research efforts have been invested in both runtime attacks as well as defenses over the past two decades. Prominent defense approaches are control-flow integrity [3, 69], code-pointer integrity [85], data-flow integrity [26], and (fine-grained) code randomization [86]. However, these solutions enforce security policies such as control-flow or data-flow integrity, without or with only limited context-sensitivity. Moreover, fine-grained enforcement of indirect forward branches remains a challenge because it is difficult to exhaustively and accurately derive the policies, e.g., the set of valid indirect destination addresses for a given branch source instruction, and to enforce them with minimal performance overhead. Generally, these approaches fail to capture and provide information about the complete state of a program's execution (which is required in detecting non-control-data attacks) and cannot mitigate DOP attacks without incurring prohibitively high performance overhead [64, 26].

Alternatively, *remote attestation* is a security service that is often deployed to allow a trusted party, called the *verifier* to establish trust in a potentially compromised and untrusted embedded device, called the *prover* by statically verifying that the program code initially loaded onto the device is unmodified. It is implemented as a challenge-response protocol where the verifier sends a challenge to the prover, and the prover in turn sends back an authenticated report to the verifier. The verifier usually generates this report by issuing a digital signature or cryptographic MAC (Message Authentication Code) over the verifier's challenge and the measurement (typically a hash computation) of the binary code that ought to be attested. However, conventional attestation schemes are static in nature, i.e., they only ensure the integrity of the program binary (that it has not been modified). They cannot provide any guarantees with respect to the *execution behavior* of this program, e.g., they cannot capture and report how the program executes, and thus cannot detect the aforementioned runtime attacks that hijack the control or data flow of execution [127] without modifying the program binary.

Hence, a software-based *runtime* remote attestation mechanism was first proposed that can capture and detect control-flow attacks [4]. The application runs in the *normal* untrusted world in a TEE while the attestation software is trusted and deployed in the *secure* world. However, being software-based in design and implementation implicates two major limitations that prohibit its practical deployment. Firstly, in order to detect control-flow events, the application code must be heavily instrumented prior to deployment. Non-instrumented or incorrectly-instrumented software cannot be attested. The instrumentation rewrites all control-flow instructions (e.g., branch, return, etc.) in the source code with trampoline instructions that capture the control-flow event and transfer it to the attestation software. This increases code size and contributes to the incurred

performance overheads. Secondly, the attestation software runs on the main processor along with the application being attested, which incurs prohibitively high performance penalties because single control-flow instructions are essentially replaced with relatively many numbers of instructions in order to track and record the control-flow event (e.g., update a running hash value). The attestation overhead increases linearly with the number of control-flow events, which means it can grow significantly for some code samples and is entirely application-dependent. Besides trampolines, the context switching between the normal and secure worlds of the TEE for the measurement to be performed in a trusted environment contributes to the overall incurred performance overhead, as well as the context switching for the hash computations themselves. Finally, the scheme assumes that the attestation software itself and its deployment in the secure world is trusted, and thus the attestation report generated can be trusted.

C-FLAT's runtime attestation scheme can only report the control flow of the execution. While it can detect some non-control-data attacks, e.g., corruption that influences the number of loop iterations, it cannot detect more sophisticated data-oriented attacks that leave the control flow still valid. Moreover, it still only mitigates attacks by *detecting* them, but does not *prevent* them altogether by enforcing a set of provided policies, for example. On the other hand, enforcement schemes can only enforce the policies provided, and thus only detect attacks that involve a deviation from these policies. Thus, they are only as good as the policies derived, whereas analyzing code (by means of static and dynamic analysis) to generate these policies exhaustively remains a challenge especially as the application code size scales. Moreover, they cannot report the overall execution behavior of the application.

While in some deployment settings, a non-intrusive tracking of program execution is desired, in others a strict enforcement of policies is necessary. Furthermore, in different deployment settings different security guarantees under different adversarial assumptions may be desired. However, at the time of our work and publications [37, 144, 38, 105, 40] (Appendices A, B, C, D, and E), according to our knowledge, no consolidated mechanism actually existed that can be *configured* flexibly within the platform at run-time to mitigate different classes of attacks and thwart different adversarial capabilities, and thus be *customized* according to the desired security/functionality requirements and deployment environment. This is particularly a challenge for potentially deploying hardwired hardware-assisted security extensions which cannot be upgraded or updated after fabrication (in contrast to software), and will thus always provide fixed security guarantees and assume the same adversarial capabilities once produced. This makes it impractical for system architects to deploy hardware-assisted mechanisms in embedded platforms, despite their advantages over software-based defenses. Despite being necessary especially in certain adversarial settings, these protection mechanisms are often entirely missing from some systems such as timing-critical real-time systems. This is usually because fail-safe operation that adheres to hard deadlines is a critical requirement of these systems, while these protection mechanisms [3, 69, 85, 4] incur non-negligible performance overheads. While this can be tolerated to some extent for applications without real-time constraints, it would violate the functionality requirements of real-time high-availability systems.

2.2 CONTRIBUTIONS

This thesis has significantly contributed to the problems described above with the following five publications that can be found in Appendices A, B, C, D, and E:

> [37] 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 IEEE/ACM Design Automation Conference (DAC). ACM, 2017. Core Rank A. Appendix A.
>
> [144] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. **ATRIUM: Runtime Attestation Resilient under Memory Attacks.** In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2017. Core Rank A. Appendix B.
>
> [38] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. **LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution**. In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2018. Core Rank A. Appendix C.
>
> [105] 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 IEEE/ACM Design Automation Conference (DAC). ACM/IEEE, 2019. Core Rank A. Appendix D.
>
> [40] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, and Ahmad-Reza Sadeghi. **CHASE: A Configurable Hardware-Assisted Security Extension for Real-Time Systems**. In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2019. Core Rank A. Appendix E.

Through our work we investigate and show how the aforementioned limitations and deficiencies of software-based security mechanisms can be overcome by relying on hardware instead. We present LO-FAT [37] (Appendix A), the first hardware-based mechanism for control-flow attestation. LO-FAT works by leveraging existing processor hardware features that inherently keep track of execution in a cycle-accurate manner. LO-FAT hardware extensions non-invasively interface with these existing processor hardware features to capture the metadata required to track execution and compute a hash measurement over the it. The computed values are communicated securely to a trusted third party to verify the control flow of the execution. This hardware-based approach enables significantly more efficient control-flow attestation in contrast with the software-based scheme, while incurring only negligible additional hardware and without requiring software instrumentation. Moreover, it provides stronger security guarantees while relying on a significantly smaller TCB than the software-based counterpart, as we show with our work and proof-of-concept implementation based on a RISC-V SoC in [37] (Appendix A).

The security guarantees of LO-FAT (as well as conventional static remote attestation schemes) rely on the assumption that attacks are software-only and that the program

code cannot be modified at runtime. In practice, these assumptions may not hold where a stronger, yet still realistic, adversary is capable of controlling and modifying the code memory such that benign code is attested but malicious code is executed, thus bypassing the attestation mechanism in place and leaving the device vulnerable to Time of Check Time of Use (TOCTOU) attacks. In our work [144] (Appendix B), we demonstrate such TOCTOU attacks on recently proposed attestation schemes by exploiting physical access to the device's memory and showing how these attacks are, in fact, realistic. To mitigate them, we present a hardware-based security extension, called ATRIUM, that provides runtime remote attestation that securely attests both the code's binary and its execution behavior, similarly to LO-FAT. However, it captures and measures the entire execution at a finer granularity and with different mechanisms such that it can also mitigate these memory manipulation attacks that are possible within a stronger adversary model. We show in [144] (Appendix B) a proof-of-concept implementation of ATRIUM on a RISC-V SoC and show how it provides resilience against both software- and hardware-based TOCTOU attacks, while incurring minimal area and performance overhead.

Both of these schemes, however, as well as C-FLAT remain vulnerable to the more sophisticated data-oriented programming (DOP) attacks [64]. Such attacks subvert these defense schemes by principally keeping the control flow and the binary of the code unmodified. They allow the adversary to execute Turing-complete malicious execution by carefully corrupting only non-control data to stitch a sequence of operations on attacker-controlled input. Prominent defenses approaches, e.g., control-flow integrity (CFI) [3, 69], code-pointer integrity [85], and (fine-grained) code randomization [86] to name some, fall short in mitigating these sophisticated attacks. These usually enforce security policies such as control-flow integrity which cannot capture non-control-data attacks which do not modify the control flow of execution. They do not provide any information about the complete state of a program's execution (e.g., required in detecting non-control-data attacks) nor can they mitigate DOP attacks without generating prohibitively high performance overhead [64].

In our next work [38] (Appendix C), we investigate how we can provide an efficient hardware-based remote attestation mechanism for RISC-based devices, called LiteHAX, that can additionally detect non-control-data attacks. LiteHAX allows to securely, efficiently and continuously capture and record both the control- and data-flow events of a program executing on a remote device and report them to a trusted verifying party. All known and reported non-control-data and DOP attacks essentially boil down to corrupting memory access operations, without inflicting any unintended control flow [64]. On RISC-V based systems, which is the target architecture in this work, memory accesses are only possible via load and store instructions. LiteHAX works, in principle, by interfacing with the processor pipeline and extracting and capturing all metadata on the execution of these memory access instructions at run-time directly from the processor in parallel to the actual execution. Therefore, similar to our earlier work, LiteHAX is minimally invasive to the processor implementation; it does not require modifications to the processor micro-architecture, neither does it require extensions to the instruction set architecture, or instrumentation of the program code. We implemented and evaluated LiteHAX on a RISC-V System-on-Chip (SoC) and show in our publication that it

incurs minimal performance and area overhead while detecting non-control-data attacks as required.

While runtime capturing and attestation of memory access operations, as shown in LiteHAX [38] (Appendix C), is one hardware-based mitigation approach to detect DOP attacks after their occurrence, run-time *enforcement* of certain policies or constraints is another promising approach that blocks the attacks before they even occur. In another work [105] (Appendix D), we propose run-time scope enforcement to efficiently mitigate all currently known DOP attacks by identifying the lexical scope rules of variables at compile time and extracting memory safety constraints from them and enforcing them at run time. To prototype our scheme, we presented HardScope [105] (Appendix D), a hardware-assisted run-time scope enforcement scheme for RISC-V based systems, that provides fine-grained context-specific memory isolation within programs. HardScope requires that the compiler is modified to instrument the program code with special instructions that record which variables may be used by each code block. Thus, HardScope requires an instruction set extension for this purpose. At run time, these instructions are used to create the different memory access rules dynamically for each individual function invocation (assuming function-level granularity) and these rules are then stored in hardware-protected stack memory. The HardScope hardware extension captures every memory access directly from the pipeline and mediates it to check it against the stored access control rules. We show in our work how HardScope can significantly reduce the exposure to data-oriented attacks with a minimal performance overhead of 3.2% for embedded benchmarks.

The different hardware-based security mechanisms we have presented above, besides other state-of-the-art approaches proposed in academia and adopted in industry, either apply enforcement or execution tracking/attestation. Moreover, each assume a different adversary model and different adversarial capabilities, thus mitigating only specific classes of attacks. No consolidated defenses exist that can mitigate multiple classes of different attack vectors, or can be even *configured* flexibly within the platform at run time to thwart different adversarial capabilities depending on the desired security/functionality requirements and deployment environment. This is particularly a challenge for these hardwired hardware-assisted security extensions which cannot be upgraded or updated after fabrication (in contrast to software), and will thus always provide fixed security guarantees and assume the same adversarial capabilities once produced. This makes it impractical for system architects to deploy these hardware-assisted mechanisms in embedded platforms, despite their advantages over software-based defenses. Secondly, timing-critical real-time systems often lack these protection mechanisms, despite. This is usually because fail-safe operation with hard deadlines is critical, while protection mechanisms, such as control-flow integrity and attestation, to mitigate or detect such attacks have been shown to incur non-negligible performance overheads. While this can be tolerated to some extent for applications without real-time constraints, it would violate the functionality requirements of real-time high-availability systems.

In our work [40] (Appendix E), we present and discuss these insights and challenges in more detail, and present a consolidated runtime-configurable security extension, called CHASE. CHASE can be more flexibly adapted to provide different security guarantees

and services at run time, e.g., either enforcement or more detailed execution tracking and attestation, depending on the desired security guarantees and the system real-time, availability and functionality requirements. This enables the adoption of such hardware-based security extensions and their customization at run time to calibrate the security vs. performance trade-off for individual use cases and deployment settings. We analyze CHASE's effectiveness in providing different security guarantees and services against different adversarial capabilities and for different use cases (e.g., real-time applications), and evaluate how this is possible with reasonable hardware logic overhead and minimal performance overhead.

3

SECURE MICROARCHITECTURE FOR TRUSTED EXECUTION

3.1 PROBLEM STATEMENT AND MOTIVATION

Modern multi-core processors are augmented with various performance optimization features that make them vulnerable to a wide spectrum of different microarchitectural attacks, as shown in recent works [83, 2, 90, 84, 81, 95, 143, 54, 48, 47, 88, 6, 5, 129, 101, 132, 134, 25, 27, 118, 20]. Shared cache resources are one of the most popular optimization features, and also the most exploited in these attacks. The inherent timing difference between a cache hit and a cache miss in shared cache behavior, while being precisely why caches are used, is also how they are exploited to infer information on the victim's execution patterns, ultimately leaking private information such as a secret key. The root cause for these attacks is mutually distrusting processes sharing the cache entries by means of deterministic and consistent set-associative access and eviction. We present next a brief overview of such cache side-channel attacks that are relevant for our work in 3.1.1, followed by the shortcomings of recently proposed defenses that our work aims to overcome in 3.1.2. Next, we focus on the encompassing security architecture itself, where we briefly discuss how state-of-the-art Trusted Execution Environment (TEE) security architectures suffer from severe shortcomings that hinder their secure and flexible deployment as desired for different emerging use cases.

3.1.1 *Cache Side-Channel Attacks*

We briefly introduce here recent cache side-channel attacks that are relevant for our work. Cache side-channel attacks have been shown to constitute a profound threat, while also playing a critical role in mounting some of the more popular attacks such as Spectre [83] and Meltdown [90]. Different types of these attacks have been demonstrated on all platforms and architectures, ranging from mobile and embedded devices [89] to server computing systems [92, 71, 147]. Furthermore, they have also been shown to undermine the promised isolation guarantees of trusted execution environments, like Intel SGX [18, 119, 100, 52] and ARM TrustZone [89, 145]. By means of these attacks, both fine-grained as well as coarse-grained private data and operations can be inferred, e.g., by bypassing address space layout randomization (ASLR) [56, 53], inferring keystroke behavior [57, 55], or leaking privacy-sensitive human genome indexing computation [18], or leaking RSA [147, 92] and AES [16, 71] decryption keys.

The attacks usually work by provoking controlled evictions of the victim's cache line, where the inherent information leakage from the access timing difference between cache hits and misses can be exploited by the adversary. We can classify the attacks into four main classes, as follows.

Access-based Approaches. In access-based attacks, e.g., Flush + Reload [60, 142], Flush + Flush [57], Invalidate + Transfer [72], and Flush + Prefetch [56], the adversary accesses the target addresses directly by flushing them out of the cache using the *clflush* instruction [1] or even exploiting timing leakage from the execution of the *clflush* instruction [57]. Flushing a target address invalidates the corresponding cache lines and writes it back to memory. Evict + Reload [55] attacks have also been shown which do not require the *clflush* instruction, but instead evict specific cache sets by accessing physically congruent addresses. These attacks are only feasible in case of shared memory between the adversary and victim, usually in the form of shared libraries, and thus, shared cache lines, and are usually effectively easier to defend against.

Conflict-based Approaches. Stealthier conflict/contention-based attacks, such as, Prime + Probe [107, 71, 75, 92, 140], Prime + Abort [45], Evict + Time [53, 107], alias-driven attacks [59], and indirect Memory Management Unit (MMU)-based cache attacks [133], require that the adversary constructs a minimal eviction set, i.e., a set of memory addresses to map to the same cache set as the target address that the adversary wants to monitor, and uses it to trigger and exploit a controlled cache contention in the same cache set of the target addresses, thus, evicting the corresponding victim cache lines. This is possible by different techniques. The adversary either measures the overall time needed by the victim process to perform certain computations [12, 16], or probes the cache with eviction sets and profiles cache activity to deduce which memory addresses were accessed [92, 71, 140, 142, 75], or accesses target memory addresses and measures the timing of these individual accesses [107, 60]. Alternatively, the adversary can also read values of addresses from the main memory to see whether cache lines that contain cacheable target addresses have been evicted to memory [59]. These attacks represent the most challenging class of attacks to sufficiently mitigate, owing to their continuously evolving sophistication and stealthiness.

Collision-based Approaches. Cache-collision timing attacks exploit cache collisions that the victim has to experience due to its own cache utilization, e.g., after a sequence of lookups performed by a table-driven software implementation of an encryption scheme, such as AES [16]. Here, the adversary is assumed capable of timing the computation of the victim process. Collision-based attacks, however, are not very commonly shown and are very specific to certain software implementations, and thus do not represent a sufficiently significant threat. The only effective architectural defense for them is locking relevant cache lines after pre-loading them.

Occupancy-based Approaches. Cache-occupancy attacks are possible in any cache architecture where adversary and victim processes compete for shared cache resources, i.e., when no strict partitioning is enforced. In these attacks the adversary observes when an eviction of his own line occurs, even if he cannot infer the address of the line that replaced it. Thus, the adversary can measure the number of evictions, use this information to infer the size of the victim's working set, and use this as a signature. A recent

attack [122] leverages this side channel to infer which website is opened in a browser tab.

3.1.2 *Shortcomings of Recent Cache Defenses*

Various types of defenses have been proposed to mitigate the aforementioned side-channel attacks, with a particular focus on the more challenging and powerful access-based and conflict-based attacks. We summarize below the different defense categories.

Side-channel Resilient Software Implementation. This aims at implementing algorithms, e.g. cryptographic algorithms, in a time-constant (thus side-channel-resilient) fashion [70, 13]. Time-constant algorithms and their implementations are not generalizable, i.e., they are hardware platform-dependent [30] and require considerable manual effort, and thus, do not represent a scalable solution.

Attack Detection. Other approaches aim to detect attacks in progress by observing hardware performance counters (e.g., on cache miss rates) [29, 109] and inferring heuristically whether an attack is underway, and consequently killing the suspicious process. However, like any heuristics-based approach, the attacks can only be discovered with a certain probability with no solid protection guarantees possible. Moreover, some stealthy variants of the attacks have been shown to not cause abnormal cache behavior [57] and would thus slip undetected through such mechanisms.

Noise Injection. Another class 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 [108, 110, 97], injecting noise into the system [135, 66] or deliberately slowing down the system clock [65, 96]. Such defenses are not fool-proof, since they do not address the fundamental root cause of the attacks, but instead only debilitate the mechanisms and means required to mount them. Moreover, they directly impact access to features, e.g., timers that are required–as actually intended–for benign functionalities. In fact, workarounds have been shown to synthesize timers still [117] or to perform attacks without relying on timers altogether [46]. Additionally, such defenses cannot protect TEE (Trusted Execution Environment) architectures, where a strong adversary capable of compromising the OS kernel is assumed, and can therefore circumvent such restrictions and still access the timers.

Cache Micro-architectural/Architectural Defenses. The approaches most related to our work are defenses which tackle the side-channel problem directly where it originates, at the cache. These defenses fall under one of two paradigms: 1) randomization-based defenses that rely on either randomized mapping tables [137, 94, 91] or cryptographic primitives Trilla18,Qureshi18,ceaser-s,scattercache,phantomcache to generate reproducible randomized mapping of memory addresses to cache sets, in order to make the attacks computationally impractical or 2) cache partitioning of any form to provide strict isolation, thus eliminating interference altogether [50, 79, 145, 93, 33, 58, 148, 76,

139, 87, 11, 136, 82, 137]. We describe the state of the art in cache side-channel defenses and their deficiencies in more detail in our work [41] (Appendix F) and [44] (Appendix G), and provide a brief summary below.

Randomization-based defenses cannot provide comprehensive and solid future-proof security guarantees, e.g., subsequent advances in minimal eviction set construction techniques have been shown to already undermine recent randomization-based defenses [116, 17, 113, 114]. In other words, these defenses usually provide security guarantees on par with the state of the art in attack algorithms/techniques, and are quickly rendered ineffective once a novel attack technique that undermines them is discovered. Once these defenses are customized to mitigate more advanced attack algorithms, e.g., more frequent re-keying of the indexing function [116], they impose prohibitively high performance overheads. Moreover, many of them rely on weak cryptographic primitives which have been shown vulnerable to cryptoanalysis, whereas deploying more secure primitives would further degrade performance and prohibitively increase hardware overheads [15, 114]. They are usually also designed to mitigate only certain classes of attacks, leaving them still vulnerable to either other attack variants or other side-channel attacks. In short, defenses in this category fail to provide well-grounded security guarantees because they do not fundamentally address the root cause for these attacks, namely, mutually distrusting code sharing cache resources.

On the other hand, cache partitioning defenses provide strict cache isolation and the desirable explicit non-interference between mutually distrusting processes, which allows to give well-justified and solid security guarantees on side-channel protection. However, existing partitioning defenses suffer from significant performance degradation, restrictive and inflexible cache utilization [137], coarse-grained allocation of the cache resources, and their inability to scale with a larger number of protection domains [136, 82, 58], as required for TEE security architectures for example. Several approaches do not directly cater for the use of shared libraries [50, 136], are architecture-specific [76, 139] or do not defend against occupancy-based attacks.

Most importantly, all of these defenses apply their side-channel cache protection for the entire execution workload impacting overall system performance, which is in practice not even required in most scenarios. They do not allow the possibility to selectively and flexibly configure the mitigation only for the security-critical portion of the workload, and thus being able to fine-tune the security vs. performance trade-off for different portions of the workload as desired.

Another approach to mitigating these side-channel attacks is flushing cache resources on every context switch to a sensitive application, as proposed by various TEE architectures [33, 19, 87, 11]. However, this does not scale well for larger caches since the performance cost incurred would become unreasonable. Moreover, flushing is not possible on a shared last-level cache, since it is shared simultaneously among multiple cores and thus, more sophisticated mechanisms are required to prevent cross-core side-channel attacks.

3.1.3 *Limitations of Existing TEE Security Architectures*

Security architectures that provide Trusted Execution Environments (TEE) protect against a privileged software adversary, e.g., a compromised operating system, by enabling the execution of sensitive services (both code and data) in isolated containers or compartments, also called enclaves. TEE architectures have been proposed for a variety of computing platforms, though we focus in our work specifically on high-performance computing platforms, e.g., industry solutions such as Intel SGX [68, 32], AMD SEV [74], and ARM TrustZone [7] or academic solutions such as Sanctum [33], Sanctuary [19], Keystone [87], and Komodo [49].

These TEE architectures usually provide only one type of enclave with the same privileges and boundaries, and thus it is required that applications which require enclave execution are adapted to the features and limitations of the enclave that the platform provides, e.g., Intel SGX restricts system calls of its enclaves and thus, applications need to be modified when being ported to SGX which incurs both deployment and performance overhead costs. In practice, it is definitely more desirable and practical if unmodified applications can be deployed directly to enclaves, and the enclave privilege level and boundaries can be specified and configured on-demand to accommodate the use case in question.

Moreover, an increasing number and variety of services now are processing sensitive/security-critical data, e.g., payment services, biometric authentication, smart contracts, speech processing, Machine Learning as a Service (MLaaS), among many others. These services are of different nature, and thus impose different requirements, both in terms of functionality and security, on the underlying TEE architecture. One such requirement, for instance, concerns the ability to establish secure, exclusive and practical binding between specific enclaves and different input/output peripherals on-demand. On some devices, for instance, privacy-sensitive data is constantly being collected over various audio, video or biometric data sensors. On devices running machine learning services, massive amounts of potentially sensitive data are often aggregated and usually offloaded to external hardware accelerators, e.g., FPGAs and GPUs, to train proprietary machine learning models. However, architectures such as SGX, SEV and Sanctum do not provide secure input/output capabilities altogether, while Keystone would require additional hardware mechanisms incorporated in order to support Direct Memory Access (DMA)-capable peripherals (e.g., GPUs and FPGAs), and other architectures would require hardware changes to the peripheral itself, e.g., to the GPU, which is only possible by the vendor itself. TrustZone, Sanctuary, and Komodo cannot even bind peripherals directly to individual enclaves. Moreover, an increasing number and variety of services now are processing sensitive/security-critical data, e.g., payment services, biometric authentication, smart contracts, speech processing, Machine Learning as a Service (MLaaS), among many others. These services are of different nature, and thus impose different requirements, both in terms of functionality and security, on the underlying TEE architecture. One such requirement, for instance, concerns the ability to establish secure, exclusive and practical binding between specific enclaves and different input/output peripherals on-demand. On some devices, for instance, privacy-sensitive data is con-

stantly being collected over various audio, video or biometric data sensors. On devices running machine learning services, massive amounts of potentially sensitive data are often aggregated and usually offloaded to external hardware accelerators, e.g., FPGAs and GPUs, to train proprietary machine learning models. However, architectures such as SGX, SEV and Sanctum do not provide secure input/output capabilities altogether, while Keystone would require additional hardware mechanisms incorporated in order to support Direct Memory Access (DMA)-capable peripherals (e.g., GPUs and FPGAs), and other architectures would require hardware changes to the peripheral itself, e.g., to the GPU, which is only possible by the vendor itself. TrustZone, Sanctuary, and Komodo cannot even bind peripherals directly to individual enclaves.

Another increasingly important requirement desired from TEE architectures is providing applications in enclaves with an adequate, practical and configurable protection against side-channel attacks, e.g., OS controlled side-channel attacks as well as cache side-channel attacks which we discussed earlier. Current industry-standard TEE architectures, e.g., SGX and TrustZone, do not consider cache side-channel attacks within their threat model altogether. Current academic architectures, such as Sanctum, propose impractical mitigation mechanisms, which would heavily degrade the OS's performance. Others, such as SEV, do not consider controlled side-channel attacks. The significant impact of these sophisticated attacks on platform security, especially cache side-channel attacks as discussed above, has already been sufficiently demonstrated, rendering them too critical to remain out of the threat model in TEE architectures. Furthermore, platform mechanisms that provide, by design, both this side-channel resilience while preserving performance and flexibility as desired for every individual application, are currently entirely missing in TEE security architectures. We elaborate on the related TEE architectures and their relevant shortcomings in more detail in our work [11] (Appendix H).

3.2 CONTRIBUTIONS

This thesis has significantly contributed to the problems described above with the following three publications that can be found in Appendices F, G, and H:

> [41] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. **HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments**. In USENIX Security. USENIX Association, 2020. Core Rank A*. Appendix F.
>
> [44] Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, and Emmanuel Stapf. **Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures**. In Annual Network and Distributed System Security Symposium (NDSS), 2022. Core Rank A*. Appendix G.
>
> [11] 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. USENIX Association, 2021. Core Rank A*. Appendix H.

To address the limitations of existing cache designs discussed above, and provide a configurable and flexible side-channel-resilient cache microarchitecture for security architectures, we propose a flexible and soft partitioning of set-associative caches and propose a hybrid cache architecture, called HybCache [41] (Appendix F). HybCache can be configured to selectively apply side-channel-resilient cache behavior only for isolated execution domains that require this sophisticated security guarantee, while providing the non-isolated execution with conventional cache behavior, capacity and performance. An isolation domain is defined as any form of compartmentalization of the workload, and can include one or more processes, specific portions of code, or a Trusted Execution Environment (e.g., SGX or TrustZone). We show in our work how, with minimal hardware modifications and kernel support, HybCache can provide side-channel-resilient cache only for isolated execution with a performance overhead of 3.5-5%, while incurring no performance overhead for the remaining execution workload. To evaluate the overhead incurred by our new microarchitecture design, we implement HybCache into an architectural simulator, gem5, and evaluate the performance overheads for the SPEC2006 benchmarks. We also provide a hardware implementation of HybCache to evaluate its hardware footprint (area and storage/memory overhead), and show through our security analysis how HybCache mitigates typical access-based and contention-based cache attacks.

While HybCache enables configurable cache side-channel resilience while maintaining non-degraded performance for the non-isolated execution, it still does not fundamentally mitigate all side-channel leakage, since it does not provide strict partitioning by design. The cache occupancy side channel, where the adversary can attempt to infer the working set size of the victim, is the only side-channel leakage that is not mitigated by the HybCache construction. This leakage is inherently available in any cache architecture where the attacker and the victim processes compete for entries in shared cache resources. It can only be effectively blocked by strict cache partitioning, which we deliberately do not provide in the HybCache construction. This allows different isolation domains to still compete for cache entries, thus preserving dynamic cache utilization for the entire workload and unaffected performance for non-isolated execution.

In a follow-up work [44] (Appendix G), we propose another cache microarchitecture design, Chunked-Cache, that blocks this cache occupancy leakage by providing strict cache partitioning thus providing clean isolation, while still maintaining flexible cache utilization. Chunked-Cache enables an execution context to "carve" out its exclusive cache chunk of configurable capacity only if it requires cache side-channel resilience. When side-channel resilience is not required, mainstream cache resources can be freely utilized. This addresses the security-performance trade-off by efficiently enabling on-demand cache side-channel resilience, i.e. only when actually required, while providing well-grounded future-proof security guarantees. Chunked-Cache provides side channel-resilient cache utilization for sensitive execution, while incurring no performance overhead on the OS due to its design mechanisms. Through our proof-of-concept implementation (on a cycle-accurate architectural simulator and a hardware implementation) and its evaluation, we show how it outperforms way-based partitioning signifi-

cantly in terms of performance and scalability for minimal hardware (logic and memory) overhead.

Our work in secure cache designs has enabled more flexible and configurable cache-based side-channel security that can be adapted on-demand for different portions of the execution workload individually and independently. To enable further configurability and flexibility for trusted execution capabilities generally, we focus next on the encompassing security architecture itself. Security architectures providing Trusted Execution Environments (TEEs) aim to protect sensitive services by compartmentalizing them in isolated execution contexts, called enclaves. However, existing TEE solutions suffer from critical shortcomings with respect to both security and functionality. They adopt a rigid approach where only a single enclave type is available, although, in fact, more flexibility is required, since different services require different types of enclaves that can adapt to the demands of the service in question. Moreover, they cannot even efficiently support emerging applications, e.g., machine learning services, which require secure binding and interaction of specific enclaves with specific peripherals (e.g., accelerators), or the computational power of multiple cores securely. Finally, their protection mechanisms against side-channel attacks, e.g., cache side-channel attacks, are either an afterthought "hotfix" or impractical for flexible usage, e.g., fine-grained allocation of cache resources to individual enclaves is usually not supported by default.

We investigate and highlight these shortcomings and challenges in our work [11] (Appendix H), and propose CURE, the first security architecture, which addresses these design goals by providing different types of enclaves whose boundaries can be flexibly configured and resources can be selectively allocated to them. Supported enclaves in CURE can either provide isolation either vertically within any single execution privilege level (sub-space enclave), or across multiple privilege levels (kernel-space enclaves) or only for unprivileged applications (user-space enclaves). In doing so, CURE already outperforms the state of the art (at time of writing) in TEEs which usually provide only one type of enclave, as stated earlier. CURE also allows that system resources, e.g., peripherals, CPU cores, or cache resources are exclusively and selectively assigned to single enclaves, thus providing the desirable fine-grained resource allocation as well as on-demand and flexible side-channel protection.

In [11] (Appendix H), we describe in detail the design challenges therein and how we tackle them in our design for CURE to successfully fulfill individual and unique functionality and security requirements of different services on demand. Besides the software stack implementation and modifications, we introduce novel hardware security primitives for the CPU cores, system bus and the shared cache in order to achieve the envisioned design goals, i.e., to adapt adequately to satisfy the different functionality and security requirements of different services. We ensure that the hardware modifications are not invasive and are also not architecture-agnostic, and can thus be ported to other platforms and architectures. While we attempt to keep the modifications in hardware reasonably minimal, we also ensure that the performance overhead for managing the enclaves in software is minimized, hence successfully achieving a reasonable trade-off in the available hardware-software co-design space.

For proof of concept, we implement CURE for the RISC-V platform using the open-source Rocket Chip generator [9]. For the fine-grained cache allocation to enclaves and cache side-channel protection, we design and implement a way-based flexible cache partitioning for the shared L2 (last-level) cache in our prototype, which we describe in more detail in [11] (Appendix H). We also emphasize that this particular cache-based partitioning/side-channel protection mechanism was selected only for convenient prototyping, whereas CURE supports that more sophisticated cache designs, such as our work HybCache [41] (Appendix F) can be easily integrated into it, as shown in [11]. We evaluate the hardware and software components of our CURE prototype in terms of the additional hardware logic and lines of software code, and show that even with minimal hardware changes, CURE can already significantly improve the state of the art of hardware-assisted security architectures. We also evaluate CURE's performance overhead on an FPGA and cycle-accurate simulator setup using micro- and macrobenchmarks, and show that CURE incurs a geometric mean performance overhead of 15.33% on standard benchmarks.

4

HARDWARE IMPLEMENTATION SECURITY

4.1 PROBLEM STATEMENT AND MOTIVATION

The gap between hardware and software security analysis is taking its toll on the security of our computing platforms. The recent outbreak of microarchitectural attacks such as Spectre and Meltdown among many others [83, 2, 90, 84, 81, 95, 143, 54, 48, 47, 88, 6, 5, 129, 101, 132, 134, 25, 27, 118, 20] has revealed how our computing platforms and hardware-based security solutions are flawed in their fundamental trust assumptions. The conventional threat model always assumed a software-only adversary and software-only vulnerabilities with the spotlight largely on software and processor architecture. However, these increasingly sophisticated attacks have been shown to exploit both software and underlying hardware flaws to compromise our computing platforms [143, 101, 80, 129, 90, 132, 54, 48], and they have emphasized to researchers and practitioners how underneath the architecture is a very complex microarchitecture and its hardware implementation that were always unjustifiably assumed trusted and secure. These attacks may trigger physical hardware effects that induce system faults or exploit microarchitectural/architectural flaws by software means to generate certain controlled microarchitectural states. Moreover, these physical or microarchitectural flaws and effects are made visible to software adversaries by means of software vulnerabilities, thus also enabling a software-only adversary to exploit these hardware vulnerabilities remotely. Platforms ranging from low-end embedded devices to complex servers, that are augmented with advanced defenses, such as data-execution prevention and control-flow integrity, have been shown vulnerable. This is because these state-of-the-art defenses aim to mitigate attacks that exploit software vulnerabilities, such as memory corruption. Furthermore, hardware-based security extensions, such as the schemes presented in Chapter 2, also aim to mitigate software attacks. They cannot mitigate attacks that exploit microarchitectural or hardware flaws. In fact, their implementation is actually vulnerable to potential hardware flaws that may not be detected at design-time, where these flaws may break the security claims of these schemes altogether. Even industry-standard security architectures, such as SGX and TrustZone, have been targets of successful microarchitectural attacks [18, 119, 100, 52, 89, 145]. Architectures proposed in academia [33, 19, 87, 11], while equipped with side-channel protection mechanisms to overcome the deficiencies of SGX and TrustZone, are not verified at the hardware implementation level to ensure that they indeed provide the claimed security guarantees.

4.1.1 *Hardware Implementation Flaws*

Hardware and System-on-Chip (SoC) designs are typically implemented at register-transfer level (RTL) by engineers using hardware description languages (HDLs), such as

Verilog and VHDL, which are in turn synthesized into a lower-level netlist representation using automated tools. At this pre-silicon design-time phase (prior to final tape-out and fabrication), hardware vulnerabilities can occur due to: (a) incorrect or ambiguous or incorrectly described/formalized security specifications, (b) flawed design, (c) flawed implementation of the design, or (d) a combination thereof. Hardware implementation bugs can be introduced either through human error by the hardware developers or by faulty compilation/synthesis of the design into its gate-level equivalent.

Even seemingly minor flaws in the implementation of a hardware module within our processor can compromise the SoC security objectives and result in denial of service, IP leakage, or exposure of assets and secrets to untrusted entities. Flaws in the underlying hardware, which serves as the foothold of our computing platforms and their security and trust assumptions, would subvert the security of all that sits above. It becomes even more critical if these flaws were committed in the implementation of hardware-based extensions dedicated to provide software security services, such as the extensions we propose in our work in Chapter 2.

The permanence of these flaws further aggravates the dilemma. Unlike software flaws, hardware vulnerabilities committed at design-time cannot be generally patched (at the actual root cause) once the hardware is fabricated. While existing industry SoCs may support microcode patching, this is only limited to a handful of changes to the instruction set architecture, e.g., modifying the interface of individual complex instructions and adding or removing instructions. These patches are firmware-only, and tend to be symptomatic fixes that circumvent the actual RTL flaw, without fundamentally patching it, while also usually incurring a performance penalty that can be avoided if the underlying problem were discovered and fixed at design-time. Besides, some vulnerabilities cannot even be patched by microcode, such as the recent Spoiler attack [73], and they require fundamentally fixing the hardware which is impossible for a legacy system.

Therefore, hardware security testing for detecting these flaws at design-time prior to fabrication in legacy systems is even more crucial than the more established software security testing.

4.1.2 *Detecting Hardware Flaws*

The semiconductor industry leverages a variety of techniques, such as simulation, emulation, and formal verification to detect these flaws.Some examples of industry tools that are leveraged for both functional as well as security-specific verification are Incisive [21], Solidify [10], Questa Simulation and Questa Formal [98], OneSpin 360 solutions [123], and JasperGold [22]. While knowledge and techniques for software security are well established both in academia and industry (e.g., regarding software exploitation and automatic bug detection techniques), security-centric HDL analysis, in comparison, lags critically behind [78, 106]. Hence, inspired by software practices [63], the chip design industry has recently adopted a security development lifecycle (SDL) for hardware [126]. This process deploys different techniques and tools, such as RTL manual code audits, assertion-based testing, dynamic verification (e.g., simulation), and automated formal verification to detect bugs in hardware designs at the pre-silicon phase, i.e., prior to

tape-out and fabrication. However, recent sophisticated cross-layer microarchitectural attacks [83, 2, 90, 84, 81, 95, 143, 54, 48, 47, 88, 6, 5, 129, 101, 132, 134, 25, 27, 118, 20] pose difficult challenges for these security verification techniques, and indicate how such stealthy bugs would still slip through these security verification processes. This is because these attacks usually exploit complex and subtle interactions between hardware and software, though existing verification techniques are fundamentally limited in modeling and verifying these interactions. They also fail to seamlessly capture some specific semantics that are relevant to many vulnerabilities, e.g., timing flow, side channels, and cache states. Generally, SDL practices are tedious, complex, largely non-automated, and require extensive human expert intervention. The correct security specifications and test cases must be exhaustively anticipated, identified, and accurately and adequately expressed using security properties and invariants that can be captured and verified by the tools.

At one end, dynamic verification techniques, e.g., simulation, involve driving a Design Under Test (DUT) with input sequences (either crafted or randomly generated) during simulation, and comparing the DUT's behavior with a set of invariants or golden reference. Such techniques are effective in identifying flaws in complex and large designs and scaling well, however, they fail to achieve deep coverage of the design's state space, and cannot uncover complex flaws.

At the opposite end of the spectrum, formal verification involves proving/disproving properties or proving the absence of an information flow of a DUT using mathematical reasoning like model checking. In contrast to dynamic verification, formal verification is capable of detecting more complex flaws but they fail, in practice, to scale to real-world, complex and large designs. To alleviate this state explosion problem, techniques such as "black-box" abstraction of a selected set of hardware modules of the design, state space constraining, and bounded-model checking are often used. However, these do not eliminate the fundamental problem and rely on interactive human expertise and manual intervention. Erroneously applying them may result in false negatives and missed vulnerabilities.

We elaborate in more detail on the SDL process and the limitations of the state-of-the-art hardware security verification in our work [39] (Appendix I).

4.2 CONTRIBUTIONS

This thesis has significantly contributed to the problems described above with the following publication that can be found in Appendix I:

> [39] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, and Ahmad-Reza Sadeghi. **HardFails: Insights into Software-Exploitable Hardware Bugs**. In USENIX Security. USENIX Association, 2019. Core Rank A*. Appendix I.

In our work, we take a closer look into the design and security assurance lifecycle of hardware, and focus the spotlight on the limitations and challenges of state-of-the-art

hardware security verification discussed above. However, this is challenging to achieve, since hardware designs with such real-world bugs are usually closed-source and proprietary, therefore it is not trivial to acquire either real-world hardware designs or real-world hardware bugs therein. Thus, the first step in qualitatively assessing the effectiveness of existing verification techniques was to construct the test harness itself, i.e., the System-on-Chip (SoC) design and the bugs therein. Together with our industry partners and collaborators at Intel, we systematically constructed a varied set of 31 hardware register transfer-level (RTL) bugs inspired from their first-hand experience with bugs that they have encountered themselves at Intel, as well as public Common Vulnerabilities and Exposures (CVEs) [103, 104, 102, 90, 83, 84] and real-world errata [62]. We injected the bugs into two open-source real-world RISC-V-based SoC designs, Pulpino [111] and PULPissimo [112]. We organized the first edition of what is now the largest international hardware security competition, Hack@DAC [131], in 2018 where 54 teams of researchers competed for three months to detect these bugs in the SoCs. While a number of the bugs could not be detected by any of the teams, several teams also detected new bugs that already existed in the open-source SoCs, which we did not inject ourselves and had no prior knowledge of. The teams largely relied on manual RTL code inspection and simulation techniques to detect the bugs. In industry, however, these are usually complemented by automated tool-based and formal verification approaches.

Thus, we conducted a second in-house investigation ourselves, in which we focused on two state-of-the-art formal verification tools (Formal Property Verification (FPV) [24] and JasperGold's Security Path Verification (SPV) [23]) to assess their effectiveness in detecting these bugs and their ease of use and friendliness. These represent the state of the art in hardware security verification and are used widely by the semiconductor industry [8], including Intel. FPV exhaustively verifies that a set of specified security properties hold true for the given RTL of a hardware design. If a security property is violation, the tool generates a counter-example, which we examine to ensure whether the property is indeed violated or if this is a false alarm. SPV leverages formal verification and path sensitization to check for illegal/unauthorized information flows.

Both the results of the competition and our investigation with formal verification tools have revealed that certain properties of RTL bugs can make them significantly more challenging to detect. With formal verification techniques, technical and practical challenges arise when attempting to scale them efficiently for larger SoCs thus requiring error-prone workaround mechanisms, such as black-box abstraction. They remain non-automated and require a certain capacity of human expertise in hardware design and intervention. They are also incapable of modeling and capturing side channels and other flows, such as timing flows as well as non-register states, e.g., cache states. Even when formal verification techniques are aided with manual inspection and simulation methods by human experts, some classes of bugs may still slip through the security analysis altogether, e.g., bugs that arise from complex and cross-modular interactions in SoCs and span multiple modules. What further aggravates the threat arising from such bugs, is the fact that many of them are in fact exploitable from software, and thus can compromise the entire SoC platform remotely, as we also demonstrate in our work. Building on our findings from both our investigation and the competition results, we attempt to

systematically classify and identify these bugs that are more challenging to detect and the characteristics that they have in common, where we call such bugs HARDFAILS.

Our work is thus the first, to the best of our knowledge, that attempts to provide a systematic and in-depth analysis of state-of-the-art hardware verification approaches for security-relevant RTL bugs. Our findings reveal qualitatively the actual capacity and effectiveness of these tools, while identifying types of bugs that are particularly more challenging to detect: HARDFAILS. We demonstrate reproducibly how these bugs can slip through current hardware security verification processes and the gravity of the security threat they pose, by showcasing how one such bug can be exploited to compromise the entire platform. We also reveal that one of the most significant and open challenges in hardware security analysis is practically anticipating and identifying all the security properties that are required in a real-world scenario, as well as specifying/formally defining them correctly. In other words, besides the technical limitations of the formal verification tools themselves, they are still only as good as the security properties that the human expert specifies and defines for the tools. There is no automated way to determine whether a tool is proving the actually intended properties, and there is no fool-proof automated approach at generating these properties, though there is some active research in this direction [146].

Ultimately, our work and insights manifest why further research is urgently required to improve state-of-the-art security verification and analysis of hardware, and sheds light on potentially promising directions, e.g., hybrid techniques that combine both formal verification and simulation-based testing that would scale better than formal verification only, as well as more efficient testing inputs generation techniques, such as fuzzing. Fuzzing is an established automated software testing technique that provides different types of data, e.g., invalid, unexpected, or entirely random data, as input to a piece of software, aka the fuzz target, and monitors if the software crashes or triggers provided code assertions, thus detecting memory corruption bugs that would otherwise be much more difficult to find out. Coverage-guided fuzzing uses program instrumentation to trace the code coverage reached by each input fed to a fuzz target. Fuzzing engines can then use this information to guide the generation of the subsequent inputs provided to the fuzz target, in order to maximize coverage. We are currently investigating whether and how fuzzing can be ported to hardware design testing, and whether it would enable more efficient coverage of the hardware DUT, e.g., better and/or faster coverage of state space and state transitions, in contrast to directed random testing.

Ever since it was first launched in 2018, we have been organizing Hack@DAC every year, and organized its first USENIX Security sequel, Hack@Sec, in 2020 [131]. Over the past few years, the competition has been growing in sophistication, size and popularity among both academics and industry professionals. Over the years, the focus of the competitions has also shifted and evolved from only bug detection and root cause analysis in 2018 and 2019 to more interestingly tooling, automation and proof-of-concept exploitation in Hack@DAC and Hack@Sec 2020. Moreover, the complexity of the deployed SoCs has also evolved, from an SoC that runs only bare-metal applications to a more complex SoC with an MMU and multiple privilege levels and a full multi-level cache subsystem and can run a small kernel. This implied that we could integrate more interesting se-

curity features such as more firmware and boot flows, cryptographic units self-tests at bootup, firmware encryption, stack canaries, thus making room for more challenging bugs as well as bugs that can span both the hardware and software of the platform. The calibre and interest of the participating teams have also evolved over the years, and so have the results and our insights. We have been involved in more exchange with some of the teams on developing and customizing open-source tools to detect some specific classes of bugs, while other teams could come up with automated exploit generation techniques. All in all, our experience with Hack@DAC and Hack@Sec over the past 3 years has given us solid insights into how bugs can be introduced in hardware designs, the varying complexity of these bugs, their security impact, and most importantly the the state-of-the-art hardware security analysis techniques. We observe how the open-source space severely lacks security analysis techniques that specifically target hardware designs, and experience first-hand the practical limitations of state-of-the-art industry-grade techniques. Plenty of open opportunities exist for developing new techniques that can address the growing challenges of analyzing the security of our hardware as it continuously evolves in size and complexity.

5

CONCLUSION

The recent surge of microarchitectural attacks has fueled a growing interest, in both academia and industry, to question the trust assumptions in the underlying hardware of our systems. These attacks have revealed the threatening consequences of hardware/microarchitecture security flaws to the entire platform security, and pressingly urge for a rethink of our hardware design paradigm where security is a key metric.

To this end, we investigate, in this thesis, the opportunities and implications of hardware-based security that emerge across the full stack of our computing platforms. Our work contributes significantly to the state of the art on multiple fronts, as we summarize next.

5.1 SUMMARY OF CONTRIBUTIONS

In Chapter 2 based on [37] (Appendix A), [144] (Appendix B), [38] (Appendix C), [105] (Appendix D), and [40] (Appendix E), we propose a suite of different hardware-based processor extensions that aim to provide dedicated security services, e.g., execution tracking, runtime attestation and policy enforcement, for security architectures. Through our work, we illustrate how the advantages of hardware can be leveraged to provide significantly more efficient security services to defend the software against different software attacks, particularly runtime memory corruption attacks. We further consolidate these different security extensions into one flexible scheme that can be configured to provide different security services or flavors, thus catering to the different security and performance requirements imposed by different applications and deployment settings.

In Chapter 3 based on [41] (Appendix F), [44] (Appendix G) and [11] (Appendix H), we investigate how our computing platforms and even dedicated hardware-based security mechanisms, similar to our work above, can be entirely compromised by attacks that exploit microarchitectural/hardware design flaws. We focus specifically on cache-based side-channel leakage, since it plays a key role in most microarchitectural attacks to date. We present two sophisticated secure cache microarchitecture designs that attempt to mitigate these attacks fundamentally by enabling secure cache resource sharing among different security domains, while still preserving performance. Our cache designs provide configurable side-channel resilience by design, i.e., the sophisticated side-channel resilience is only enabled when desired, thus providing a flexible performance-security calibration that can adapt to different applications. We further extend this flexibility to trusted execution environment (TEE) architectures themselves, and propose the first security architecture, which can provide different types of enclaves whose boundaries can be configured and system resources, including cache resources, can be selectively and exclusively allocated to them.

In Chapter 4 based on [39] (Appendix I), we delve one layer beneath the microarchitecture and design, and scrutinize the actual implementation of the hardware. Through extensive case studies and our Hack@DAC hardware security competition, we examine how the actual implementation of the hardware can also harbor RTL vulnerabilities. We survey the design and security assurance lifecycle of hardware, and focus the spotlight on the limitations and challenges of state-of-the-art hardware security verification. Through the results of the competition and our investigation with formal verification tools, we show how certain properties of RTL bugs can make them significantly challenging to detect, both by manual inspection as well as formal verification techniques. Our work and insights indicate that further research is pressingly required to improve state-of-the-art security verification and analysis of hardware, and sheds light on potentially promising directions.

5.2 FUTURE WORK AND OUTLOOK

To address the open challenges in hardware and microarchitectural security fundamentally and reinstate trust in the underlying hardware of our computing platforms, we envision a radically different design paradigm fo a security-adaptive platform that can sustainably serve secure next-generation computing platforms. Such a platform would aim to provide more consolidated and comprehensive full-stack security-aware and adaptive primitives which span both the software and hardware layers of the platform. These cross-layer primitives would be integrated by pro-active design into the computing platform architecturally and micro-architecturally to address the performance-security trade-off fundamentally and flexibly. Ultimately, these primitives would then be configured on-demand by means of an interfacing configuration engine to 1) adapt to customized requirements with regards to performance, compatibility and security for different use cases, and to 2) adapt to different adversarial settings and mitigate emerging threats.

Figure 3 demonstrates our vision for such a platform where adaptive hardware elements are integrated within the microarchitecture and configured in different flavors. They can be *tightly* and *invasively* integrated within individual key components and features of the CPU/SoC, e.g., different cache mapping and partitioning mechanisms that can be configured on-demand to selectively provide side-channel-resilient cache. On the other hand, a less invasive and more coarse-grained flavor of integration would involve adaptive hardware extensions that only *interface* with the CPU/SoC to provide hardware-based security services to the software, e.g., control-flow integrity or runtime attestation of the executing software (in line with our work [37, 144, 38, 105, 40] (Appendices A-E). These different security primitives, being ingrained fundamentally in the hardware with varying degrees of integration and granularities, allow the platform to efficiently adapt to changing security/performance requirements. They can be configured to enforce different security policies through a secure interfacing configuration engine as shown in Figure 3. Ultimately, this would also equip vendors with more degrees of freedom and customization in updating platform security policies and applying full-

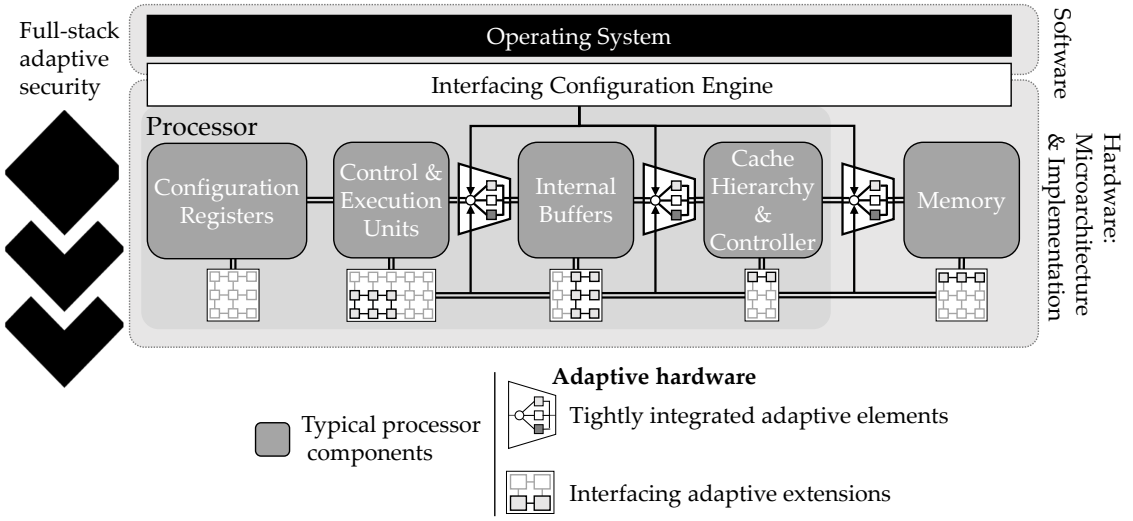


Figure 3: Our vision for future computing platforms that incorporate adaptive hardware primitives to achieve full-stack configurable security

stack security patches (across both the hardware, microcode and software) to mitigate the latest threats, where currently this is limited to software/microcode patches only.

Designing and developing this platform opens up multiple concrete challenges and directions for future research. Mechanisms are required to provide hardware-based security extensions that interface with the CPU/SoC such that they can be very flexibly configured to provide a spectrum of different security services that can adapt to emerging threats and different deployment/adversarial settings. Moreover, the hardware-software co-design space for these extensions needs to be exhaustively investigated to identify the sweet spot that would combine the best of both hardware and software worlds. This would also enable that they can sufficiently scale as required, e.g., to either support more complex software applications or instead only secure the TCB of the computing platform. To increase synergy, techniques to further extend these services to leverage them for different purposes when necessary, e.g., hardware-assisted software fuzzing (which also relies fundamentally on software execution tracking), are also required.

Practically integrating adaptive hardware elements invasively within the processor/-SoC microarchitecture also poses different interesting challenges. Mechanisms to achieve this efficiently, while still providing flexible configuration, are required. Furthermore, enabling this configurability (to calibrate the security-performance trade-off) at the different microarchitectural features and units (and not only caches which has been the focus in this thesis), while still preserving the desired performance benefits, is another research challenge.

Finally, scalable techniques that can efficiently verify and analyze the security properties and specifications of both the design and implementation of these extensions and hardware primitives, as well as their interactions with the overlying software are essential.

On a final note, envisioning and developing such a platform with security-aware hardware design primitives and investigating the full hardware-software co-design space has only recently become both viable and valuable owing to the advent of open hardware, such as RISC-V and the emergence of open-source RISC-V processor and SoC implementations. This has also coincided with the outbreak of microarchitectural attacks, where both of which have evolved the focus in system security in the last few years, shifting the spotlight to the long ignored security implications of hardware and the unjustified trust assumptions therein. New territories and opportunities have emerged where the security of hardware microarchitecture, design and implementation can be better scrutinized, making room for the state of hardware security to eventually catch up with that of software security.

BIBLIOGRAPHY

[1] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2016.

[2] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html, 2018.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009. ISSN 1094-9224. doi: 10.1145/1609956.1609960. URL http://doi.acm.org/10.1145/1609956.1609960.

[4] 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 SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[5] 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.

[6] 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.

[7] 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.

[8] Robert Armstrong, Ratish Punnoose, Matthew Wong, and Jackson Mayo. Survey of Existing Tools for Formal Verification. Sandia National Laboratories https://prod.sandia.gov/techlib-noauth/access-control.cgi/2014/1420533.pdf, 2014.

[9] 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.

[10] Averant. Solidify. http://www.averant.com/storage/documents/Solidify.pdf, 2018.

[11] 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*. USENIX Association, 2021.

[12] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[13] 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.

[14] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[15] 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.

[16] Joseph Bonneau and Ilya Mironov. Cache-collision Timing Attacks Against AES. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2006.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Cadence. Incisive Enterprise Simulator. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html, 2014.

[22] Cadence. JasperGold Formal Verification Platform. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-

verification/formal-and-static-verification/jasper-gold-verification-platform.html, 2014.

[23] Cadence. JasperGold Security Path Verification App. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html, 2018. Last accessed on 09/09/18.

[24] Cadence. JasperGold Formal Property Verification App. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html, 2021.

[25] 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.

[26] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298470.

[27] 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.

[28] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251398.1251410.

[29] 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.

[30] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In *CCS*, 2014.

[31] Computerworld. Stuxnet renews power grid security concerns. http://www.computerworld.com/article/2519574/security0/stuxnet-renews-power-grid-security-concerns.html, 2010.

[32] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical report, Cryptology ePrint Archive. Report 2016/086, 2016. https://eprint.iacr.org/2016/086.pdf.

[33] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*, 2016.

[34] Ivan De Oliveira Nunes, Ghada Dessouky, Ahmad Ibrahim, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. Towards Systematic Design of Collective Remote Attestation Protocols. In *International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[35] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated Synthesis of Optimized Circuits for Secure Computation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[36] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the Communication Barrier in Secure Computation using Lookup Tables. In *Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2017.

[37] 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 *IEEE/ACM Design Automation Conference (DAC)*. ACM, 2017.

[38] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution. In *IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2018.

[39] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, and Ahmad-Reza Sadeghi. HardFails: Insights into Software-Exploitable Hardware Bugs. In *USENIX Security*. USENIX Association, 2019.

[40] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, and Ahmad-Reza Sadeghi. CHASE: A Configurable Hardware-Assisted Security Extension for Real-Time Systems. In *IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2019.

[41] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *USENIX Security*. USENIX Association, 2020.

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

[43] Ghada Dessouky, Ahmad-Reza Sadeghi, and Shaza Zeitouni. SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

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

[45] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-free High-precision L3 Cache Attack Using Intel TSX. In *USENIX Security*, 2017.

[46] 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.

[47] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[48] 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.

[49] 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.

[50] Michael Godfrey. On The Prevention of Cache-Based Side-Channel Attacks in a Cloud Environment. Master's thesis, Queen's University, Ontario, Canada, 2013.

[51] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC'87*, pages 218–229. ACM, 1987.

[52] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, 2017.

[53] 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.

[54] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*, 2018.

[55] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security*, 2015.

[56] 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.

[57] 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.

[58] 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*. USENIX Association, 2017.

[59] 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.

[60] 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.

[61] Hewlett-Packard. Data Execution Prevention. http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf, 2006.

[62] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[63] Michael Howard and Steve Lipner. The Security Development Lifecycle. *Microsoft Press Redmond*, 2006.

[64] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 969–986, Washington, DC, USA, 2016. IEEE Computer Society. doi: 10.1109/SP.2016.62. URL http://doi.ieeecomputersociety.org/10.1109/SP.2016.62.

[65] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[66] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[67] Imperva. Breaking Down Mirai: An IoT DDoS Botnet Analysis. https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html, 2016.

[68] Intel. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[69] Intel. Control-flow Enforcement Technology Preview, 2016. URL https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[70] Intel. Intel Integrated Performance Primitives Cryptography Developer Reference. 2019.

[71] 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.

[72] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2016.

[73] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security*. USENIX Association, 2019.

[74] Kaplan et al. AMD memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[75] 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.

[76] 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.

[77] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. VoLTpwn: Attacking x86 Processor Integrity from Software. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[78] Hareesh Khattri, Narasimha Kumar V Mangipudi, and Salvador Mandujano. Hsdl: A security development lifecycle for hardware technologies. *IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 116–121, 2012.

[79] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security*. USENIX Association, 2012.

[80] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur 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.

[81] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[82] 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.

[83] 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.

[84] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Security*, 2018.

[85] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685061.

[86] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 276–291, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.25. URL http://dx.doi.org/10.1109/SP.2014.25.

[87] 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.

[88] 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.

[89] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security*, 2016.

[90] 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.

[91] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

[92] 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.

[93] 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.

[94] 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.

[95] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[96] 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.

[97] 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.

[98] Mentor. Questa Verification Solution. https://www.mentor.com/products/fv/questa-verification-platform, 2018.

[99] Dover Microsystems. Your processor needs a bodyguard. https://www.dovermicrosystems.com/, 2021.

[100] 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.

[101] 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.

[102] NIST. Broadcom Wi-Fi chips denial of service. https://nvd.nist.gov/vuln/detail/CVE-2012-2619, 2012.

[103] NIST. AMD: Backdoors in security co-processor ASIC. https://nvd.nist.gov/vuln/detail/CVE-2018-8935, 2018.

[104] NIST. AMD: EPYC server processors have insufficient access control for protected memory regions. https://nvd.nist.gov/vuln/detail/CVE-2018-8934, 2018.

[105] 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 *IEEE/ACM Design Automation Conference (DAC)*. ACM/IEEE, 2019.

[106] Jason Oberg. Secure Development Lifecycle for Hardware Becomes an Imperative. https://www.eetimes.com/author.asp?section_id=36&doc_id=1332962, 2018.

[107] 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.

[108] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[109] Mathias Payer. Hexpads: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[110] Colin Percival. Cache missing for fun and profit, 2005.

[111] PULP Platform. Pulpino. https://github.com/pulp-platform/pulpino, 2018.

[112] PULP Platform. Pulpissimo. https://github.com/pulp-platform/pulpissimo, 2018.

[113] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic prime+ probe attacks and covert channels in scattercache. *arXiv preprint arXiv:1908.03383*, 2019.

[114] 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.

[115] Moinuddin K. Qureshi. Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

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

[117] 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.

[118] 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.

[119] 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.

[120] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[121] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315313. URL http://doi.acm.org/10.1145/1315245.1315313.

[122] 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.

[123] OneSpin Solutions. OneSpin 360. https://www.onespin.com/fileadmin/user_upload/pdf/datasheet_dv_web.pdf, 2013.

[124] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2015.

[125] Ebrahim M. Songhori, Thomas Schneider, Shaza Zeitouni, Ahmad-Reza Sadeghi, Ghada Dessouky, and Farinaz Koushanfar. GarbledCPU: A MIPS Processor for Secure Computation in Hardware. In *IEEE/ACM Design Automation Conference (DAC)*. IEEE Press, 2016.

[126] Sunny .L He and Natalie H. Roe and Evan C. L. Wood and Noel Nachtigal and Jovana Helms. Model of the Product Development Lifecycle. https://prod.sandia.gov/techlib-noauth/access-control.cgi/2015/159022.pdf, 2015.

[127] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *IEEE Symposium on Security & Privacy (IEEE S&P)*. IEEE Computer Society, 2013.

[128] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *NDSS*, 2020.

[129] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *USENIX Security*, 2017.

[130] 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.

[131] Intel TU Darmstadt, Texas A&M University. HACKEVENT. https://hackatevent.org/, 2021.

[132] 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.

[133] 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*, 2018.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security*, 2019.

[139] 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.

[140] 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. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2019.

[141] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS'86*, pages 162–167. IEEE, 1986.

[142] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security*, 2014.

[143] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[144] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. ATRIUM: Runtime Attestation Resilient under Memory Attacks. In *IEEE/ACM International Conference on Computer-Aided Design*. IEEE, 2017.

[145] 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.

[146] Rui Zhang and Cynthia Sturton. Transys: Leveraging common security properties across hardware designs. In *IEEE Symposium on Security & Privacy (IEEE S&P)*, 2020.

[147] 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.

[148] 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.

A

LO-FAT: LOW-OVERHEAD CONTROL FLOW ATTESTATION IN HARDWARE

[37] 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 IEEE/ACM Design Automation Conference (DAC). ACM, 2017. Core Rank A.

LO-FAT: Low-Overhead Control Flow ATtestation in Hardware

Ghada Dessouky[1], Shaza Zeitouni[1], Thomas Nyman[2,3], Andrew Paverd[2], Lucas Davi[4], Patrick Koeberl[5], N. Asokan[2], Ahmad-Reza Sadeghi[1]

[1]Technische Universität Darmstadt, Germany
{ghada.dessouky,shaza.zeitouni,ahmad.sadeghi}@trust.tu-darmstadt.de

[2]Aalto University, Finland
thomas.nyman@aalto.fi, andrew.paverd@ieee.org, asokan@acm.org

[3]Trustonic, Finland
thomas.nyman@trustonic.com

[4]University of Duisburg-Essen, Germany
lucas.davi@uni-due.de

[5]Intel Labs, Germany
patrick.koeberl@intel.com

ABSTRACT

Attacks targeting software on embedded systems are becoming increasingly prevalent. Remote attestation is a mechanism that allows establishing trust in embedded devices. However, existing attestation schemes are either static and cannot detect control-flow attacks, or require instrumentation of software incurring high performance overheads. To overcome these limitations, we present LO-FAT, the first *practical hardware-based* approach to control-flow attestation. By leveraging existing processor hardware features and commonly-used IP blocks, our approach enables efficient control-flow attestation without requiring software instrumentation. We show that our proof-of-concept implementation based on a RISC-V SoC incurs no processor stalls and requires reasonable area overhead.

1 Introduction

Embedded systems have been facing a variety of security challenges for decades [25] which are becoming increasingly prevalent with emerging trends such as collaborative Internet of Things (IoT). A recent prominent example is *Mirai* malware[1] in October 2016, where a series of Distributed Denial-of-Service (DDoS) attacks against the DNS system disrupted a number of prominent websites.These attacks were perpetrated by IoT devices, including routers, DVRs, and web-enabled security cameras, that had been compromised by the Mirai malware.

Increasingly, attacks against embedded systems aim to exploit software vulnerabilities. In 2015, a remotely exploitable buffer overflow vulnerability was found in the *USB over IP* software used in millions of residential gateways and wireless routers supplied by prominent manufacturers[2]. In 2014, a memory corruption flaw was found in the embedded webserver software used by over 200 different models of embedded devices, affecting at least 12 million devices, many of which still remain vulnerable today[3].

Remote attestation is an important class of security mechanisms designed to detect software attacks. In principle, remote attestation allows one entity (the *verifier*) to ascertain the precise state of the software running on a remote system (the *prover*). However, most attestation schemes are *static* in that they attest the software initially loaded by the prover before it begins executing. Although useful, this still leaves the system vulnerable to *run-time* software attacks. If the adversary gains control of the stack or heap, (s)he can alter control-flow information to subvert the control flow of the target program, and mount a *code-reuse attack*. Similarly, in *non-control data* attacks [8], the adversary modifies strategic data variables to cause a permissible but unintended control flow change (e.g., executing a privileged instruction sequence). Traditionally, code-reuse attacks are mitigated using techniques such as control-flow integrity (CFI) [1]. However, CFI cannot prevent non-control data attacks, since these do not violate control-flow integrity. Neither of these types of attacks can be detected by static attestation.

To overcome these challenges *control-flow attestation* [2] was proposed very recently, enabling the prover to precisely report the control flow of application software to the verifier while giving assurance on control-flow integrity and detection of non-control data attacks. The attestation mechanism of [2] requires an isolated execution environment (e.g., ARM TrustZone, Intel SGX) to protect it against potentially compromised application software. However, implementing control-flow attestation in software has two limitations: Firstly, in order to detect control-flow events, the application software must be *instrumented* prior to deployment. Non-instrumented or incorrectly-instrumented software cannot be attested. The instrumentation rewrites all control-flow instructions (e.g., `branch`, `return`, etc.) in order to transfer control to the attestation software. Secondly, the attestation software runs on the main processor which incurs significant performance penalties because single control-flow instructions are essentially replaced with relatively many numbers of instructions in order to track and record the control-flow event (e.g., update a running hash value). As we elaborate in §7, some existing hardware approaches, such as debugging and tracing features in modern processors [14, 24] or hardware security architectures [3, 6, 9], can be used to record control flow information. However, due to the overhead they incur or the type

[1]https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html

[2]http://blog.sec-consult.com/2015/05/kcodes-netusb-how-small-taiwanese.html

[3]http://mis.fortunecook.ie/

DAC '17, June 18 - 22, 2017, Austin, TX, USA

ISBN 978-1-4503-4927-7/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3061639.3062276

of information they record, these approaches are not well-suited for control-flow *attestation*.

Goals and Contributions. To overcome the limitations of a software solution, we introduce a practical hardware-based Low-Overhead Control Flow ATtestation architecture, LO-FAT. Unlike software implementations, LO-FAT can handle *unmodified application software* without instrumentation, meaning that it is transparent to legacy software. By recording the control flow in hardware in parallel to the main processor, LO-FAT does not stall the application software, thus eliminating the performance overhead of attestation in software. LO-FAT leverages existing processor features and commonly-used IP blocks and can feasibly be implemented on typical embedded systems hardware platforms.

The main contributions of this paper are:

- Design of LO-FAT, a hardware-based scheme for control-flow attestation, providing the *same security guarantees* as previous software schemes, without the performance overhead or the need for software instrumentation (§4).
- An integrated optimization for eliminating redundant attestation computation (e.g., avoiding duplication when attesting loops) and reducing the burden on the verifier (§4).
- A proof-of-concept implementation of LO-FAT on the new open-source RISC-V architecture targeting the Pulpino core for single-threaded embedded system software (§5).
- A systematic evaluation of LO-FAT in terms of the required hardware area and performance benefits (§6).

2 Problem Setting and Challenges

Remote attestation provides a well-known mechanism for detecting malware on a device. However, existing conventional (binary) attestation cannot detect run-time exploitation techniques, since run-time attacks do not not modify the program binary. Such attacks aim to subvert the intended control flow of the targeted program while it is executing. An overview of different classes of such attacks is shown in Figure 1. In general, a program reserves dedicated memories for data and code. The former is marked as readable and writable *(rw)*, whereas the latter is as readable and executable *(rx)*. This ensures that code cannot be executed from data memory, and code memory cannot be overwritten. Furthermore, any program can be abstracted through its corresponding control-flow graph (CFG) that encapsulates the valid paths a program should follow at run-time.

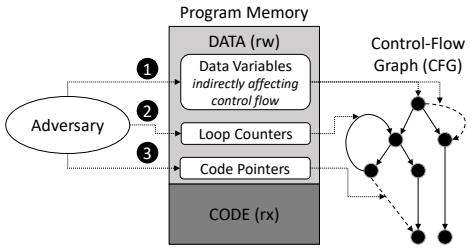


Figure 1: Overview on run-time attack classes

We can distinguish three classes of run-time attacks: ❶ non-control-data attacks that indirectly affect the control flow of a program, ❷ corruption of loop counter variables, and ❸ code-pointer overwrites. The most prominent run-time attacks exploit code-pointer overwrites, i.e., corruption of return addresses and function pointers. For instance, code-reuse attacks such as *Return-oriented Programming* (ROP) [23] exploit memory corruption vulnerabilities (e.g., buffer overflows) in the program and then stitch together a malicious sequence of machine code instructions from benign *gadgets* of code already residing in the vulnerable program memory. This is exemplified by a malicious CFG edge (see dashed line for code-pointer overwrite in Figure 1). These attacks have been shown

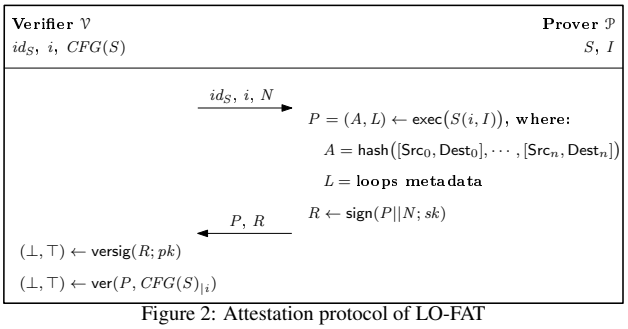


Figure 2: Attestation protocol of LO-FAT

to be a realistic threat on many processor architectures, such as Intel x86 [23], ARM [17] and embedded systems building on Atmel AVR [12]. Although countermeasures against this class of attacks exist, e.g., control-flow Integrity (CFI) [1] and code-pointer integrity (CPI) [16], they do not prevent attacks ❶ and ❷. The so-called *non-control data attacks* [8] do not compromise the control flow of a program, but cause unexpected malicious control-flow paths by corrupting data variables. In ❶, the attacker compromises data variables that are used for security decisions during program execution, e.g., corrupting an authentication variable to execute a privileged but existing path. Attack class ❷ is even more subtle as it only affects the number of times a program loop is executed. This can have severe consequences in the context of embedded system software, e.g., a syringe pump dispenses more liquid than requested (see [2]).

Control-flow attestation can cover these cases by assuring the verifier of the precise run-time control flow of the program on the embedded device. In [2], the first control-flow attestation scheme was proposed and implemented. However, it suffers from practical limitations, such as high performance overhead and the need for tedious software instrumentation.

Our work tackles the challenge of detecting attack classes ❶-❸, while addressing the limitations of recently proposed software-based control-flow attestation [2] by presenting LO-FAT, an efficient hardware-only solution.

3 System Model

Figure 2 depicts the attestation protocol of LO-FAT: the verifier $\mathcal{V}$ aims to attest the run-time control-flow (execution path) of the Program S on a remote embedded system – the prover $\mathcal{P}$. We assume that both $\mathcal{V}$ and $\mathcal{P}$ have access to the program S in binary form and that conventional static (binary) attestation assures $\mathcal{P}$ is executing the correct and unmodified program S.

First, $\mathcal{V}$ performs a one-time offline pre-processing step to generate the CFG of S (including expected loop execution information) by means of static or dynamic analysis. Next, $\mathcal{V}$ initiates the protocol by sending $\mathcal{P}$ the program input i for the program ID id_S, and the nonce N to ensure freshness of the attestation response. $\mathcal{P}$ executes S with verifier input i and a set of malicious adversary inputs I. In fact, the untrusted inputs received may corrupt the control-flow by means of the attack techniques described in §2. While S executes, LO-FAT captures the control-flow transitions and generates a cumulative authenticator A of the control-flow path taking source and destination address $(Src, Dest)$ of each branch as input. Naively storing and transmitting every single executed instruction to $\mathcal{V}$ would incur impractical memory, power and communication overheads, especially for resource-constrained embedded devices. Hence, LO-FAT follows the idea outlined in [2] and computes a cumulative cryptographic hash of the executed path. In addition, it also produces auxiliary metadata L to track program loop paths and their number of iterations (including recursive functions) thereby covering attacks of class ❷ in Figure 1. Together A and L form

a unique program path P. Lastly, upon program exit, $\mathcal{P}$ generates the *attestation report* $R = sign(P||N; sk)$, under the signing key sk, which is stored by $\mathcal{P}$ in hardware-protected secure memory, e.g., a register that is accessible only to LO-FAT. Upon receiving R, $\mathcal{V}$ verifies the signature using the verification key pk. Next, $\mathcal{V}$ checks whether the reported path P resembles a valid path in CFG under input i. If true, $\mathcal{V}$ is assured of $\mathcal{P}$'s execution.

Adversary Model and Assumptions. We assume a strong adversary that has full control over the *data memory* of $\mathcal{P}$ and can utilize standard memory corruption vulnerabilities to modify arbitrary writable memory locations. However, the adversary cannot modify program code at run-time (marked as *rx*) and cannot modify memory used by LO-FAT itself (due to hardware protection). Note that similar to all attestation schemes we consider software-only attacks and hence physical attacks on $\mathcal{P}$'s device are out of scope in this work. Also note that our scheme can detect attacks that affect the program's control-flow, but not pure data-driven attacks (that do not affect any control-flow) such as *data-oriented programming* attacks, which remain an open research problem [13].

4 LO-FAT Design

Figure 3 illustrates our architecture for LO-FAT and how it interfaces with the processor pipeline. The proposed scheme exploits branch tracking functionality inherent in any processor pipeline and re-usable IP cores such as the hash engine. We extend these with additional logic to achieve efficient tracing of control-flow information. The main LO-FAT components are the branch filter and the loop monitor. The former extracts branch instructions from the processor as it executes the attested code segment while the latter monitors program loops.

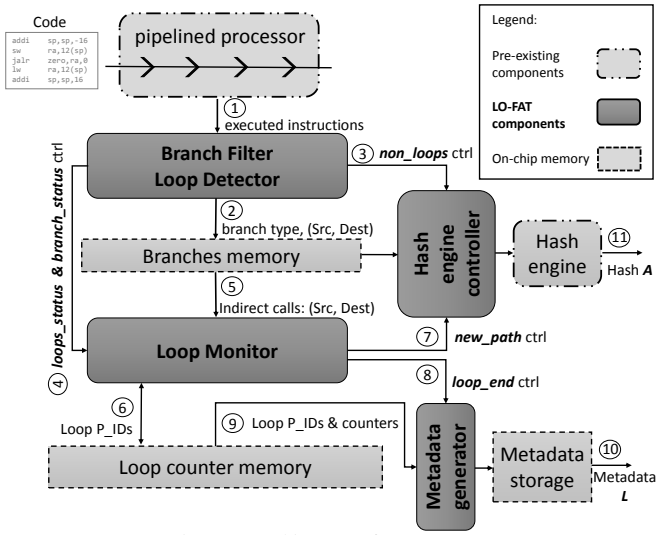


Figure 3: Architecture of LO-FAT.

Branch Filter. Upon code execution, the branch filter, which is tightly coupled to the processor, extracts the current program counter and instruction executed per clock cycle. Then it filters in every `branch`, `jump` and `return` instruction since these are the relevant instructions for control-flow attestation. The branch filter outputs a concise representation of every executed branch instruction with its source and destination address pair $(Src, Dest)$ into a dedicated branches memory and detects whether the intercepted branch is within a program loop. If not, the branch filter enables hashing of $(Src, Dest)$. Branches inside a program loop require special treatment in LO-FAT, because (i) loop counter manipulation may compromise the program's control-flow in a malicious way (§2), and (ii) naively hashing each loop iteration and path leads to a combinatorial explosion of valid hash values [2]. As such, we design LO-FAT to compress control-flow information associated with loops efficiently. As mentioned earlier in §3, we report each loop path and its number of iterations as auxiliary metadata L. However, doing so in hardware is challenging, i.e., in contrast to the most related work C-FLAT, since we do not use code instrumentation to preserve legacy compliance. Hence, the branch filter must detect and identify loop entry and exit points and their depth at run-time without instrumentation aid. We describe in §5.1 how we tackled this challenge.

Loop Monitor. When a loop is encountered, the branch filter forwards the loop entry and exit to the loop monitor. The loop monitor identifies and tracks program loops (including nested loops). When a branch inside a program loop is encountered, the branch filter forwards this information to the loop monitor which in turn encodes each path inside the loop uniquely. Simultaneously, $(Src, Dest)$ of each `branch` remains stored in the branches memory.

Another major challenge concerning loops is the hash computation and attestation overhead incurred by hashing each loop iteration. In LO-FAT, we significantly reduce the hash computation cost by only hashing each loop path once and keeping an iteration counter for each unique loop path. To achieve this, LO-FAT generates a unique path encoding for each loop path and associates an on-chip loop counter with it. The loop monitor indicates newly observed loop paths to the hash engine controller in order to hash its corresponding $(Src, Dest)$ from the branches memory. On the other hand, once the same loop path executes, LO-FAT only needs to increment the counter, i.e., not requiring further hash operations.

Upon loop exit, the loop monitor requests the metadata generator to assemble the loop auxiliary metadata based on the loops memory which contains the unique loop path encodings, their number of iterations, and indirect branch targets. This information is stored on-chip and is appended to the final hash value A computed at the end of the attested execution. Finally, a digital signature R is computed over the hash value A, metadata L and nonce N and sent to $\mathcal{V}$ for attestation (as per our protocol outlined in §3).

5 Implementation

5.1 Loop Handling

Detecting loops. As shown in Figure 3, the branch filter unit traces the instruction (and its address) executed per clock cycle and filters in (1) every `branch`, `jump` and `return` instruction. It outputs a concise representation of every executed branch instruction with its $(Src, Dest)$-pair into a dedicated branch buffer ((2)). To compress the control-flow trace for loops, the branch filter has to detect loops. If the intercepted branch is not in a loop, the branch filter sends the control signal *non_loops_ctrl* to the existing hash engine controller to compute a hash over $(Src, Dest)$ in (3). Otherwise, the branch filter forwards the loop status (entry and exit) to the loop monitor and its depth (in case of nested loops) via the *loops_status_ctrl* signals ((4)).

To enable efficient run-time loop detection, we utilize a property of RISC architectures that implement a *link-register*, such as PowerPC, ARM, SPARC, and RISC-V. LO-FAT uses a simple heuristic to differentiate between backward branches that constitute loops, and branches for subroutine calls where the call target resides earlier in memory. Since subroutine calls use instructions that update the *link-register*, we consider the target of each *non-linking* backwards branch as a *loop entry node*. The basic block proceeding the branch instruction is considered a *loop exit node*. We base our heuristic on our observations of the RISC-V compiler assembly and the calling

convention described in the instruction manual: any subroutine call with multiple call sites must be *linking* and updates the *link-register*. Subroutines with a single call site are still compiled as a *linking* branch or are optimized by traditional inlining using the RISC-V compiler.

The addresses of the entry and exit nodes of each loop are stored in registers by the loop detector and used to detect and track loop iterations and loop depth at run-time when executing nested loops. The number of loop iterations is determined by recording the number of times the loop entry node is entered within the loop. Loop termination is detected by tracking if execution proceeds to or past the currently active loop exit node, either as the result of sequential execution (e.g. in the case of a conditional branch) or a non-linking branch (e.g. break). Loop execution status is forwarded using the *loops_status_ctrl* signals to the loop monitor, as shown in Figure 3.

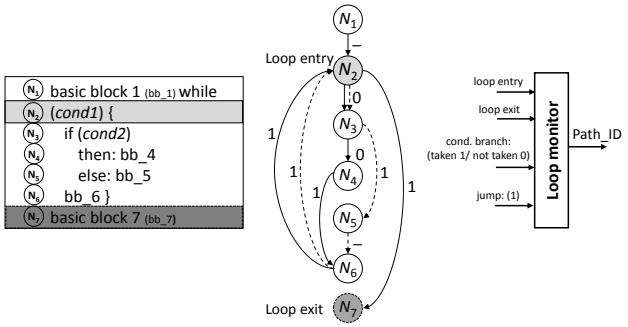


Figure 4: CFG for pseudo-code and its layout of instructions in memory.

Tracking loops. As shown in Figure 3, the loop monitor receives *branch_status_ctrl* signals from the branch filter to describe the type of intercepted branch instruction and its $(Src, Dest)$ (⑤). This branch tracking mechanism allows the loop path encoder to uniquely encode paths as they occur. Simultaneously, $(Src, Dest)$ of each `branch` along the executing loop path remain stored in the branches memory.

Figure 4 shows a sample pseudo-code and its CFG according to how the instructions would be laid out in code memory to illustrate how the loop monitor encodes the loop paths. The example code shows a *while-loop* with an *if-else* statement inside. Each basic block in the pseudo-code is represented by a node in the CFG and numbered accordingly, with loop entry and exit nodes also indicated. Within this simple loop, there are only 2 valid paths: bold path $N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6 \rightarrow N_2$ and dashed path $N_2 \rightarrow N_3 \rightarrow N_5 \rightarrow N_6 \rightarrow N_2$.

For every `conditional branch`, the processor evaluates the condition and either jumps to the computed target address (branch is taken), or continues sequentially to the next instruction address in memory (branch is not taken). Processors commonly track this branching behavior in the pipeline and may encode a taken/not-taken branch with '1'/'0'. This branch information is extracted from the processor by the branch filter and used by the loop monitor to uniquely identify and encode paths within each loop with a unique path_ID, as shown in Figure 4. In Figure 4, the dashed path $N_2 \rightarrow N_3 \rightarrow N_5 \rightarrow N_6 \rightarrow N_2$ is encoded as '011' and bold path $N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6 \rightarrow N_2$ as '0011'. Other path encodings are considered invalid and detected by the $\mathcal{V}$.

Once a loop path is completed, this unique path_ID is used to index *loop counter* memory, in which the number of iterations for each corresponding path is saved (⑥) in Figure 3. A counter value of zero indicates the first time a particular path is executed. This is forwarded by the loop monitor into the hash engine controller using *new_path_ctrl* signals (⑦) to enable hashing of corresponding $(Src, Dest)$ pairs. Otherwise, the counter is simply incremented.

To ensure constant-time, single-cycle memory access latency, we implement *loop counter memory* as on-chip memory indexed by the unique loop path encodings. However, this consumes a dedicated sparsely-utilized memory which is often a constrained resource on low-end embedded devices. In light of this, LO-FAT allows configuring the granularity of the control-flow tracking according to the availability of memory resources.

Once a loop exits, this is identified by the loop monitor and indicated in the *loop_end_ctrl* signals sent to the metadata generator (⑧). The metadata generator assembles the loop auxiliary metadata from the loops memory - this consists of the unique loop path encodings in order of first occurrence, the number of iterations of each path, and the indirect branch targets encountered in this loop (⑨). This fine-grained auxiliary information on loop execution is stored on-chip (⑩) and is appended to the final hash value computed at the end of the attested execution (⑪). Finally, a digital signature is computed over the hash value, metadata and nonce N, and sent to $\mathcal{V}$ for attestation. Handling indirect branches in loops is yet another implementation challenge we discuss next.

5.2 Handling Indirect Branches in Loops

Indirect branches can involve any arbitrary number of targets which can never be exhaustively identified using static analysis. To uniquely identify loop paths with indirect branches (calls and returns), we would need to include the 32-bit target addresses into the path encodings, which would require infeasibly high memory requirements for loop path-indexed memory. Instead, we re-encode the addresses using a smaller number of n bits, allowing a maximum number of 2^n-1 possible targets for each loop. Target addresses are encoded at run-time and stored in a register file, which is implemented as 2 interleaved CAMs to ensure low-latency constant-time access. When a target address is encountered that exceeds the configured limit, we report this in the encoding to the $\mathcal{V}$ by an all-zero code. LO-FAT is designed such that the maximum number of branches per loop path and the maximum number of possible target addresses (of indirect branches) to track is configurable in a trade-off between granularity and availability of on-chip memory. Tracking ℓ branches per path in a loop requires $8 \times 2^\ell$ bits memory. In our implementation, we configure $n = 4$ to track up to 16 possible indirect branch targets for a given loop and $\ell = 16$ such that LO-FAT can handle a maximum of 16 branches per loop path (every additional indirect branch tracked reduces the maximum number of possible conditional branches by n) and depth of up to 3 nested loops, which requires a dedicated 1.5 Mbits memory that is synthesized as block RAM (BRAM) when prototyping on FPGA. Once a loop exists, its memory is re-used for other subsequent loop executions.

Loop metadata. The measurement in A is a single hash computation of $(Src, Dest)$ pairs of executed loop paths. To enable $\mathcal{V}$ to reconstruct the final hash value, metadata L of the loops serves as helper data and provides $\mathcal{V}$ with fine-grained insight into the execution of the loops. L contains the encodings of executed paths in each loop, the order of first occurrence of each executed path, and number of iterations per loop path and indirect branch targets.

5.3 Hash Engine

A single hash measurement A is computed on the full execution path, along with auxiliary loop metadata L. We employ a SHA-3 512-bit open-source engine[4] operating at a maximum clock frequency of 150 MHz. It consists of a permutation module which operates on a message block size of 576-bit. User input is absorbed by the core first into a padding module to assemble the 576-bit block size. Once this padding is full, the permutation module begins computation on

[4] http://opencores.org/project,sha3

input. In LO-FAT, the engine can absorb a 64-bit input $(Src, Dest)$-pair every clock cycle into the padding module for 9 clock cycles, after which the 576-bit buffer becomes full and notifies the permutation module to begin its computation. Once full, the padding buffer cannot absorb further input for 3 clock cycles after which it resumes normally. Therefore, a small cache buffer is configured at the hash engine input to prevent dropping of $(Src, Dest)$-pairs if they arrive during these cycles where the padding buffer is full. Using this hash engine, an unlimited message size can be hashed while indicating the end of streaming $(Src, Dest)$-pairs when the execution of attested software is completed.

6 Evaluation

We present a proof-of-concept implementation of LO-FAT on Pulpino [18], the first open-source RISC-V-based microcontroller SoC [19]. It is based on a single 32-bit 4-stage minimal RISC-V core targeting low-end embedded systems. We augment the RISC-V processor pipeline to interface with the LO-FAT branch filter to extract control-flow signals required for execution flow tracing. LO-FAT can be easily integrated into any low-end embedded processor as it does not require modifications to the ISA.

6.1 Functionality and Performance

We integrated LO-FAT with Pulpino and performed cycle-accurate functional simulation of their RTL Verilog source code on ModelSim while Pulpino executed extracted code segments from real embedded applications, such as Open Syringe Pump[5], an open-source open-hardware syringe pump design. Simulation results confirmed the functionality of LO-FAT in correctly capturing and compressing the control flow (branches, loops, and nested loops) of an uninstrumented application. Since LO-FAT extracts and filters control-flow events in parallel with the processor, it does not incur any performance overhead for the attested software, as opposed to C-FLAT which incurs attestation overhead that is linearly dependent on the number of control-flow events. LO-FAT internally incurs latency of 2 clock cycles for branch instructions and loop status tracking and 5 clock cycles at loop exit for completing path_ID generation and *loop counter* memory access and update. However, LO-FAT simultaneously continues to absorb and process any incoming $(Src, Dest)$-pairs to prevent the processor from stalling or dropping trace information. Synthesis results using Xilinx Vivado indicate LO-FAT can operate at maximum clock frequency of 80 MHz on a Virtex-7 XC7Z020 FPGA device on a Zedboard. The LO-FAT units are engineered such that they operate on par with Pulpino's clock frequency, while also allowing single-cycle constant-time memory accesses for indirect branches and loops management. Eliminating the CAM access results in a much higher clock frequency if desired.

The length of the auxiliary metadata (L) that must be sent to $\mathcal{V}$ depends on the number of loops executed, the number of different paths per loop, and the number of indirect branch targets encountered in the attested code.

6.2 Area

On a Virtex-7 XC7Z020, LO-FAT consumes 4% of the available registers and 6% of available LUTs, which amounts to an average of 20%additional logic overhead to the Pulpino SoC. 49 36Kbit Block RAM (BRAMs) are utilized, most of which are dedicated for the sparse loop path-indexed memories to ensure constant-time single-cycle access. Therefore, its width depends on the configured maximum number of indirect branches allowed in each loop path and number of bits required to encode them, as discussed in §5.2. In our implementation, the loop monitor is configured to tackle up to 4 indirect branches and requires 10 bits to encode them in $Path\_ID$, resulting in 16 BRAMs per loop. Since we allow up to 3 levels of nested loops, we require 48 BRAMs. Configuring these parameters to lower numbers or leveraging CAMs instead reduces the memory requirements significantly at the expense of coarser granularity or additional logic overhead respectively.

[5] https://hackaday.io/project/1838-open-syringe-pump

6.3 Security

The primary security requirement of LO-FAT is to provide an *accurate*, *complete*, *authentic*, and *fresh* attestation of $\mathcal{P}$'s control flow. This requires an integrity-protected mechanism for recording control-flow information and unforgeably communicating this to $\mathcal{V}$.

Control-Flow Recording. One of the main contributions of LO-FAT is using low-overhead hardware extensions to record control-flow information preventing it from being modified or subverted by malicious software. The on-chip memory employed by LO-FAT for storing the $(Src, Dest)$ addresses prior to their hashing is also assumed to be protected from adversarial access. The hardware extensions are guaranteed to receive every control-flow event from the processor, thus ensuring that the complete control flow is recorded. All $(Src, Dest)$ addresses are cryptographically hashed resulting in the authenticator A. The auxiliary metadata L records (1) the unique paths within each loop; (2) the number of repetitions of each path; and (3) all indirect branches encountered within loops.

Attestation Protocol. LO-FAT makes use of the widely-used secure challenge-response attestation protocol. As explained in §3, $\mathcal{P}$ sends the recorded program path P along with a digital signature over P and a nonce supplied by $\mathcal{V}$. If $\mathcal{P}$'s signing key has not been compromised, this signature guarantees the authenticity of the attestation, and the inclusion of the challenge nonce ensures freshness. Our assumed software adversary cannot compromise the signing key because it is stored in hardware-protected secure memory. Any tampering with the attestation messages can be detected by $\mathcal{V}$.

Given that the control flow recording and the signing key is protected from software attacks, the resulting attestation report provided by LO-FAT is accurate, complete, authentic, and fresh. Since $\mathcal{P}$'s code is immutable and is statically attested at boot time, $\mathcal{V}$ has complete information about $\mathcal{P}$'s execution. As described in §3, $\mathcal{V}$ also has access to the CFG of the attested software, which it can use to identify permissible control flows and detect control-flow attacks or non-control data attacks.

7 Related Work

Remote Attestation. Most prior work focuses on *static* remote attestation [7, 11, 21], which is orthogonal to run-time attestation – the focus of this paper. Software-based attestation [22] can, under strict assumptions, enable static attestation of legacy devices without hardware-based trust anchors. Property-based attestation [20] can attest behavioral characteristics of a program, with the assistance of a trusted third-party. However, none of these can attest control-flow at machine code instruction level.

Prior work on run-time attestation focuses on specific aspects of a program's execution. ReDAS [15] attests program data invariants, such as the integrity of a function's base pointer, at each system call. Trusted virtual containers [4] attest the run-time launch order of application modules – a form of coarse-grained control-flow attestation that does not include internal control flows within modules. DynIMA [10] uses dynamic taint analysis and tracing to attest run-time properties that may be symptomatic of run-time attacks. However, it does not cover non-control data attacks and incurs high performance overhead due to dynamic taint analysis.

C-FLAT [2] is a fine-grained control-flow attestation scheme. LO-FAT also leverages the idea of attesting the control flow of an application by computing a cumulative hash of executed branches but with several fundamental differences. C-FLAT requires *instrumentation* of all control-flow instructions thereby violating legacy compliance. In contrast, LO-FAT does not require any binary rewriting. C-FLAT requires complete coverage in the offline binary analysis, as un-instrumented control-flow instructions could be exploited to mount undetectable attacks. This is not possible in LO-FAT as every executed branch is monitored by design. Finally, C-FLAT incurs significant performance overhead, whereas LO-FAT incurs no performance overhead due to its efficient hardware support for control-flow attestation.

Tracing and Debug Mechanisms. Intel processors provide the *Last Branch Record* (LBR) and *Branch Trace Store* (BTS) mechanisms, which can be used to trace control-flow events [24]. However, the overhead incurred by these debugging mechanisms makes them unsuitable for control-flow attestation. Recently, Intel processors introduced *Intel Processor Trace* (IPT) [14], a low-overhead execution tracing feature that collects more tracing information than BTS (including execution mode and timing information). However, IPT cannot be directly used for control-flow attestation as it only reports control-flow events that cannot be inferred from static analysis. ARM's CoreSight[6] debug and trace architecture provides a mechanism to access trace information from different hardware trace components. However, high-throughput tracing on ARM typically requires the use of proprietary hardware.

Hardware-Assisted Security. Recent work [5, 26] developed a generic architecture for enforcing a diverse range of SoC security policies. Each IP block has an individually-customized security wrapper that sends security-relevant events and information to a central security controller to enforce individual security policies for each IP. However, this incurs high memory and logic complexity overhead as the number of IPs increases. It has further been proposed [3, 6] that this could be made more practical by re-purposing design-for-debug features found on many SoCs – a promising approach which could complement LO-FAT in future.

Sofia [9] is a recent hardware-assisted architecture for enforcing control-flow integrity (CFI). It encrypts instructions with CFI-dependent data, such that they can only be decrypted at run-time as part of a valid control-flow path, and it ensures instruction integrity by checking MACs on groups of instructions at run-time. However, unlike LO-FAT, this requires software instrumentation and places decryption in the critical execution path, thus incurring total execution time overheads of up to 110%.

8 Conclusion

Due to the increasing prevalence of interconnected embedded systems, software running on these devices have become a prime target for remote attacks. We presented in this paper the first hardware-based control-flow attestation scheme that allows precise detection of remote memory corruption attacks in embedded system software. Our architecture, LO-FAT, monitors, measures and reports the program's behavior by interfacing with the processor to intercept control-flow events. LO-FAT does not require any code instrumentation (compliant to legacy software), compiler toolchain or instruction set extension. Our proof-of-concept implementation on the open-source RISC-V core is highly efficient with no performance impact on the attested software at the expense of minimal logic overhead and on-chip memory.

Acknowledgments. This work was supported by the German Science Foundation CRC 1119 CROSSING project, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant number 643964 (SUPERCLOUD), Tekes — the Finnish Funding Agency for Innovation (CloSer project), and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

[6] https://www.arm.com/products/system-ip/coresight-debug-trace

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, pages 4:1–4:40, 2009.
[2] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *ACM CCS*, 2016.
[3] J. Backer, D. Hely, and R. Karri. On Enhancing the Debug Architecture of a System-on-Chip (SoC) to Detect Software Attacks. In *IEEE DFT*, 2015.
[4] K. A. Bailey and S. W. Smith. Trusted Virtual Containers on Demand. In *ACM-CCS-STC*, 2010.
[5] A. Basak, S. Bhunia, and S. Ray. A Flexible Architecture for Systematic Implementation of SoC Security Policies. In *ACM/IEEE DAC*, 2015.
[6] A. Basak, S. Bhunia, and S. Ray. Exploiting Design-for-Debug for Flexible SoC Security Architecture. In *ACM/IEEE DAC*, 2016.
[7] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *ACM/IEEE DAC*, 2015.
[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security*, 2005.
[9] R. d. Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. d. Bosschere, B. Preneel, B. d. Sutter, and I. Verbauwhede. SOFIA: Software and Control Flow Integrity Architecture. In *ACM/IEEE DATE*, 2016.
[10] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks. In *ACM CCS-STC*, 2009.
[11] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *ISOC NDSS*, 2012.
[12] A. Francillon and C. Castelluccia. Code Injection Attacks on Harvard-architecture Devices. In *ACM CCS*, 2008.
[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On The Effectiveness of Non-Control Data Attacks. In *IEEE S&P*, 2016.
[14] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Chapter 36 Intel Processor Trace. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2016.
[15] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang. Remote attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *IEEE/IFIP DSN*, 2009.
[16] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *USENIX OSDI*, 2014.
[17] L. Le. ARM Exploitation ROPMap. BlackHat USA, 2011.
[18] Pulpino. An Open-Source Microcontroller System based on RISC-V. https://github.com/pulp-platform/pulpino.
[19] RISC-V. The Free and Open RISC Instruction Set Architecture. https://riscv.org/specifications, 2016.
[20] A.-R. Sadeghi and C. Stüble. Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms. In *NSPW*, 2004.
[21] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security*, 2004.
[22] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based Attestation for Embedded Devices. In *IEEE S&P*, 2004.
[23] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM CCS*, 2007.
[24] M. L. Soffa, K. R. Walcott, and J. Mars. Exploiting Hardware Advances for Software Testing and Debugging (NIER Track). In *ACM/IEEE ICSE*, 2011.
[25] J. Viega and H. Thompson. The State of Embedded-Device Security (Spoiler Alert: It's Bad). *IEEE S&P*, 10(5):68–70, 2012.
[26] X. Wang, Y. Zheng, A. Basak, and S. Bhunia. IIPS: Infrastructure IP for Secure SoC Design. *IEEE Transactions on Computers*, Aug 2015.

B

ATRIUM: RUNTIME ATTESTATION RESILIENT UNDER MEMORY ATTACKS

[144] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. ATRIUM: Runtime Attestation Resilient under Memory Attacks. In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2017. Core Rank A.

ATRIUM: Runtime Attestation Resilient Under Memory Attacks

Shaza Zeitouni
TU Darmstadt, Germany
shaza.zeitouni@trust.tu-darmstadt.de

Ghada Dessouky
TU Darmstadt, Germany
ghada.dessouky@trust.tu-darmstadt.de

Orlando Arias
University of Central Florida, USA
oarias@knights.ucf.edu

Dean Sullivan
University of Florida, USA
deanms@ufl.edu

Ahmad Ibrahim
TU Darmstadt, Germany
ahmad.ibrahim@trust.tu-darmstadt.de

Yier Jin
University of Florida, USA
yier.jin@ece.ufl.edu

Ahmad-Reza Sadeghi
TU Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

***Abstract*—Remote attestation is an important security service that allows a trusted party (verifier) to verify the integrity of a software running on a remote and potentially compromised device (prover). The security of existing remote attestation schemes relies on the assumption that attacks are *software-only* and that the prover's code cannot be modified at runtime. However, in practice, these schemes can be bypassed in a stronger and more realistic adversary model that is hereby capable of controlling and modifying code memory to attest benign code but execute malicious code instead – leaving the underlying system vulnerable to Time of Check Time of Use (TOCTOU) attacks.**

In this work, we first demonstrate TOCTOU attacks on recently proposed attestation schemes by exploiting physical access to prover's memory. Then we present the design and proof-of-concept implementation of ATRIUM, a *runtime* remote attestation system that securely attests both the code's binary and its execution behavior under memory attacks. ATRIUM provides resilience against both software- and hardware-based TOCTOU attacks, while incurring minimal area and performance overhead.

***Index Terms*—Attestation, runtime, memory attacks**

I. Introduction

Recent high-profile attacks on embedded systems, such as Mirai and Stuxnet, have become crucially alarming and of increased significance as systems are becoming more interconnected and collaborative. *Remote attestation* plays an important role as a security service for detecting malware on a remote device. It is implemented as a challenge-response protocol that allows a trusted *verifier* to obtain an authentic report about the (software) state of a potentially untrusted remote device called *prover*. Conventional attestation schemes are static in nature, i.e., the prover sends an authenticated report to the verifier by issuing a digital signature or cryptographic MAC (Message Authentication Code) over the verifier's challenge and the *measurement* (typically hash) of the binary code to be attested [22]. However, static attestation only ensures the integrity of binaries but *not* of their execution. In particular, it cannot detect the prevalent state-of-the-art runtime attacks that do not modify the program binary but subvert the intended control flow of the targeted application program during its execution. Current runtime attacks take advantage of code-reuse techniques, such as return-oriented programming that dynamically generate malicious code by chaining together code snippets (called gadgets) of benign code *without* requiring to inject any malicious code/instructions [24]. Consequently, the hash value computed over the binaries remain unchanged and the attestation protocol succeeds, although the prover has been compromised. These sophisticated exploitation techniques have been shown effective on many processor architectures, such as Intel x86 [23], SPARC [4], ARM [16], and Atmel AVR [10]. In fact, large-scale investigations of embedded systems security have shown various vulnerabilities, including memory corruption (such as buffer overflow) that can be exploited for runtime attacks.

Hence, effective attestation should enable reporting the prover's dynamic behavior – more concretely, its current execution details – to the verifier. To attest the dynamic program behavior researchers have proposed enhancements and/or extensions to static binary attestation (e.g., [11], [3]). The most recent, C-FLAT [3], reports the prover's dynamic state (execution paths) and provides fine-grained control-flow measurements to the verifier. Note that, unlike control-flow integrity (CFI) enforcement, control-flow attestation provides detailed information about the executed path that might be of crucial interest to a remote verifier. This information helps in detecting data-oriented non-control attacks [5] that can bypass CFI by corrupting data variables to execute a valid but unintended control-flow path, for instance, redirecting the control flow to a high-privileged recovery routine (see also [13]). However, C-FLAT requires program code instrumentation and incurs high performance overhead, particularly on the prover.

On the other hand, all existing attestation schemes (including C-FLAT) rule out physical attacks in their adversary model. This assumption is not always realistic, since the adversary may at some point have physical access to the prover. In this case, it is possible to execute (extraordinarily effective and cheap) non-invasive attacks on the program code memory through *physical access*. In particular, the adversary physically controls and modifies the memory such that benign code is attested but malicious code is executed instead.

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

Goals and Contributions. In this paper, we first demonstrate that – using external interfacing with prover's program code memory bank – an adversary can bypass all existing attestation schemes and deliver sound attestation reports, without even having to extract the prover's secret keys (cf. § III). To overcome the limitations of current attestation schemes, we introduce a holistic approach to attestation ATRIUM, a *resilient runtime attestation* scheme that is capable of detecting both physical memory attacks and software attacks including runtime attacks by attesting the executed instructions and their control flow at runtime. Our main contributions are listed as follows.

- We demonstrate memory bank attacks on state-of-the-art attestation schemes for embedded devices such as SMART [9] and C-FLAT [3]. We exploit physical access to code memory to bypass attestation and deliver sound attestation reports without having to extract the prover's secret keys.
- We present ATRIUM– an attestation scheme which: (1) detects memory bank attacks by attesting instructions as they are fetched from (off-chip) memory for execution; (2) prevents software attacks on the attestation process itself by separating the attestation engine from the processor (i.e., no instructions are sent to the processor to perform attestation). Instead, attestation is performed by a separate hardware engine in parallel. (3) detects runtime attacks by tracking and reporting both executed instructions and control-flow events during execution.
- We present a proof-of-concept implementation and performance analysis which demonstrate the effectiveness and feasibility of ATRIUM, and its applicability to low-end embedded devices.

II. Background

Control-Flow Graph (CFG). The execution flow of a program can be abstracted into a control-flow graph (CFG) by leveraging the aid of static and dynamic code analysis. The nodes in CFG represents basic blocks of a program, while edges represent control-flow transitions from one block to another by means of a branch instruction. A *valid* path in CFG is composed of several nodes connected by edges.

Runtime Attacks. An outline of the different classes of runtime attacks is illustrated in Figure 1. The system dedicates separate memories for data and code. The former is marked as readable and writable *(rw)*, while the latter is marked as readable and executable *(rx)*. This ensures that code cannot be executed from data memory, and code memory cannot be overwritten *by means of software*. Along this CFG, we can outline three major classes of runtime attacks: ❶ non-control-data attacks that indirectly affect the control flow of a program, ❷ corruption of loop variables, and ❸ code-pointer overwrite attacks. By corrupting control-flow information stored in the stack or heap and overwriting code-pointers (return addresses and function pointers) as in ❸ an attacker can redirect the control flow of a program such that execution has a malicious and unauthorized effect. In attacks based on *code-injection*,

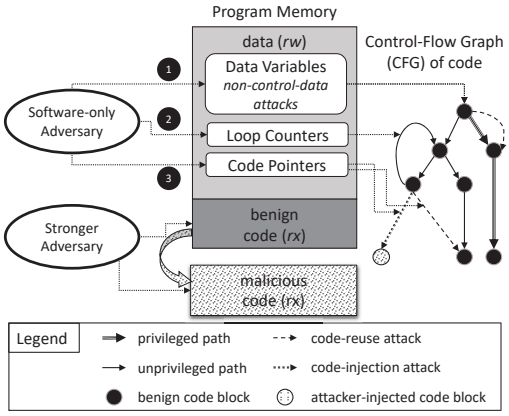


Figure 1: Different attack classes

the attacker places a malicious executable payload in program memory and redirects control flow to execute it. Alternatively, state-of-the-art runtime attacks leverage *code-reuse* techniques, such as *Return-oriented Programming* (ROP) [23]. These attacks exploit a memory corruption vulnerabilities (e.g., buffer overflows) in the program and stitch together a malicious sequence of machine code instructions from benign *gadgets* of code already residing in the memory of the vulnerable program. *Non-control-data attacks* [5] do not compromise the control flow of a program, but cause unexpected malicious control flow by corrupting critical data variables such as an authentication variable. This results in executing a privileged (unintended) but permissible control-flow path that exists in the CFG. Attack ❷ affects the number of times a program loop executes by corrupting a loop variable such as a counter. This can have severe consequences depending on the context, e.g., a syringe pump dispenses more liquid than requested (see [3]). Code injection attacks can be prevented by either marking memory as writable or executable. This mechanism is known as *Data Execution Prevention* (DEP) [12]. Countermeasures against code reuse attacks include: *Control-Flow Integrity* (CFI) [2], fine-grained code randomization [19], and Code-Pointer Integrity (CPI) [18].

Besides software-based runtime attacks, a stronger adversary as shown in Figure 1, can modify program code in memory through *physical access* without mounting sophisticated invasive physical attacks, but by simply replacing the benign code memory with malicious code memory at runtime. We elaborate on these memory bank attacks next in § III and propose an attestation scheme that can mitigate them in § V.

III. TOCTOU Attacks on Attestation Schemes

Next we describe memory bank attacks that we aim to mitigate in this work, and we show how they bypass recently proposed attestation schemes: SMART [9] C-FLAT [3] and LO-FAT [7]. These attacks assume a stronger adversary that can physically manipulate the code memory without the need for sophisticated invasive physical attacks and can consequently bypass attestation schemes that strictly consider software-only adversary. The attack is illustrated in Figure 2: At $\mathcal{P}rv$'s side

the attestation scheme (i.e., the attestation code and secret key) is stored on-chip while the benign code resides in an external memory. The adversary can interleave instruction fetches to malicious code in-between those fetches needed to attest the benign code of the original program. This can be done by replacing the original memory interface with an interface to a memory controller. This allows the adversary to direct instruction fetches to either benign code when attestation is running, or malicious code otherwise. The same interleaving attack can be achieved by inserting malicious instructions in-between hooks to the attestation. As long as the malicious instructions do not interfere with attesting benign code, e.g., intended control flow, the attestation can be bypassed. In the following, we describe how we implement the attacks to bypass SMART and C-FLAT.

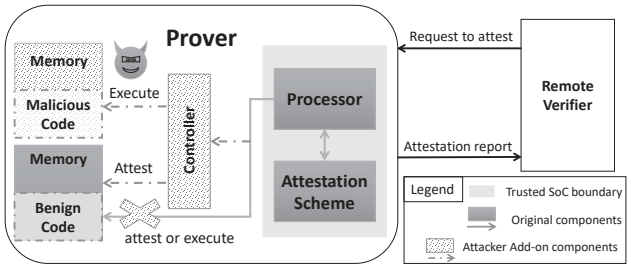


Figure 2: Memory bank attack on attestation schemes

A. SMART

SMART [9] is a static attestation scheme that establishes a root of trust in low-end embedded systems with minimal hardware components. It targets microprocessors that are able to execute code from an external memory, whereas the attestation code and key reside in an internal ROM and are protected by access control policies of a memory protection unit (MPU). When an attestation request is received, the *atomic* attestation code in ROM computes a HMAC of a region of code memory, provided in the attestation request. Then the attested code executes *atomically*.

Detecting Attestation Execution. By eavesdropping in the communication channel between the verifier and the prover for an attestation request, we determine when the attestation engine is about to run in order to launch a TOCTOU attack. Although this is permissible by the adversary model in SMART, we choose not to tackle the detection problem this way. Instead, we examine a side-channel that is inherent to the SMART design, by placing a monitor on the address bus between the processor and memory to capture which addresses are being accessed. Using the access patterns, we are able to discern whether a CPU is executing from external memory or from the internal ROM. Since SMART is prototyped on the open-source MSP430[1], it utilizes a von Neumann architecture, where data and instructions are accessed over the same address space but are structured such that they reside in different sections of memory. Hence, we can extract and filter out data accesses, leaving behind accesses to code memory. In doing so, we observe the time-frame that it takes the internal ROM to set up the attestation environment, followed by the linear scan of code addresses, then the subsequent execution of external code. On processors with modified Harvard architecture, a temporary halt in accesses to code memory would be recognized, as the ROM code starts executing. We then observe a linear scan over an address range, as code is being read and hashed by the attestation code. A break is then noticed as the ROM code cleans up memory, followed by the continued access to program memory for execution. Utilizing this, we perform one of the following attacks to mount a TOCTOU attack.

Blind Execution of Malicious Software. Since code memory remains external to the SoC, we splice the address bus, add a new memory chip containing malicious code and utilize the monitor to detect when the attestation code runs. When attesting, we bank to the memory with the intended code. When executing, we bank to the malicious code memory, allowing SMART to report valid attestation results while malicious code is actually executed by CPU during periods of no attestation.

Leakage of Secrets via Data Memory Banking. As the attestation code runs, temporary values are saved in memory, assuming SMART implementation utilizes off-chip memory to store temporary values. We use the monitor to detect when the attestation code runs. As data memory is accessed to store temporary values, we bank memories to allow for the leakage of values. We perform this by physically tampering with the address lines between the processor and the memory. As the monitor detects when SMART is about to perform its cleanup routines, we bank to a different portion of memory, leaving the ROM code to erase the wrong portion of memory. By reading the SMART secrets from memory, we are able to reconstruct the attestation secret and fake a valid response.

B. C-FLAT

C-FLAT [3] is a runtime attestation scheme that aims to measure and report the control-flow behavior of an executing code. It instruments all branch instructions such that they are intercepted by a *runtime tracer* (RTT). The RTT recovers the source and destination addresses of the branch as well as its type, which are then passed to the *measurement engine* (ME). The ME is responsible for computing a hash over the reported branches and these hash measurements are secured by running in a TrustZone secure world. In this way, a runtime control-flow attestation report is generated and verified against previously computed control-flow traces stored in a trusted verifier party.

C-FLAT is susceptible to two TOCTOU attacks assuming that the attacker has physical access to the code memory : 1) replacing instructions within a basic block with malicious ones; and 2) refactoring the control-flow graph (CFG) of an arbitrary program to match a benign CFG protected by C-FLAT. Both attacks exploit the fact that C-FLAT attests *only* control flow when exiting a basic block but not the executed instructions themselves. Hence, intermediate instructions within the basic block can be arbitrarily replaced by malicious executable code by a stronger adversary with physical access to the code

[1] http://opencores.org/project,openmsp430

memory, as long as the control flow of the code remains unchanged and the expected attestation report is not violated. These attacks are also applicable to the hardware-assisted control-flow attestation scheme LO-FAT [7] since it also only attests control flow.

We chose to implement a TOCTOU attack against one of the case studies presented in [3], namely the syringe pump program responsible for dispensing intravenous (IV) fluids. Our attack goal is to dispense liquid in incorrect volumes at unexpected times, thereby, disrupting the correct flow of IV fluids. We only demonstrate the second attack variant, however, the first variant of the attack is also easily feasible by replacing the original instructions within the basic block with malicious ones. This allows the original RTT hooks into the ME to compute a valid attestation report as it is based upon the source and destination addresses of a branch and its type.

In place of the original program that manages liquid dispensing and withdrawal, we implement a malicious program that chooses a random value to dispense by modifying the `set-quantity` function and additionally creates compound dispense and withdraw triggers for the `move-syringe` function. We embed this code in the original program, which creates new edges in the CFG of the syringe pump program. Our new edges would violate C-FLAT's attestation report for the benign syringe pump program.

To avoid triggering C-FLAT, we refactor the CFG of our attacker syringe pump program using the REpsych tool[2] to construct the desired CFG. The REpsych tool is an IDA plugin that translates a source image into a functioning program whose CFG is the image. We used the original syringe pump's CFG as a source image, and our modified syringe pump program as the target. This allowed us to generate a program with alternative functionality, but equivalent CFG to the original syringe pump program. We then recompute the attestation report using C-FLAT's tools[3]. The attacker program's attestation report matched the original syringe pump program's attestation report after CFG refactoring. Thus, we were able to accurately execute the attacker program without violating C-FLAT's protection.

IV. ATRIUM

We present ATRIUM a runtime attestation scheme targeting bare-metal embedded systems software. ATRIUM comprises a remote embedded system, called in this context the prover $\mathcal{P}rv$, and a trusted verifier $\mathcal{V}rf$. The $\mathcal{P}rv$ is deployed in-field such that the adversary has physical access to its memory. Typically, both $\mathcal{V}rf$ and $\mathcal{P}rv$ have access to the binary code of the program P to be attested on $\mathcal{P}rv$. Note that, in practice, it may not be feasible to apply runtime attestation to the entire program code due to obvious efficiency reasons, but it can be applied to pre-defined security-critical code regions.

A. Adversary Model and Assumptions

In addition to the standard capabilities of the adversary in typical remote attestation schemes, which assume software-only attacks, our adversary can also perform runtime attacks (§ II). Furthermore, we assume a stronger adversary that has physical access to the $\mathcal{P}rv$'s memory and can manipulate the program code at runtime and, therefore, is able to mount a TOCTOU attack (§ III). However, the adversary cannot modify memory reserved and used by ATRIUM itself – this memory is hardware-protected and not mapped to the software-accessible address space. *Data-oriented programming attacks* [13] that do not affect the control flow as well as invasive physical attacks on the SoC that aim at extracting secret keys are out of scope. This assumption is reasonable, since an adversary is more likely to mount a simple physical attack on the memory as we demonstrated in § III, rather than expensive sophisticated invasive attacks on the chip that can destruct it eventually.

B. Runtime Attestation: High-Level Scheme

Inspired by C-FLAT [3] (described in § III-B) and the hardware-assisted scheme LO-FAT [7], ATRIUM performs attestation of an executing program code at runtime. However, unlike both schemes, it measures both the executed instructions (to detect the more advanced TOCTOU attacks described in § III) and control flow (to detect runtime attacks).

Similar to C-FLAT, our attestation mechanism relies on $\mathcal{V}rf$ performing one-time offline pre-processing to generate the CFG of program P (including expected loop execution information) by means of static and dynamic analysis. $\mathcal{V}rf$ computes cryptographic hash measurements over the instructions and addresses of basic blocks along legal CFG paths and stores them in a reference database. $\mathcal{V}rf$ initiates the attestation by sending $\mathcal{P}rv$ benign input in_b, the code region to be attested in P, and a nonce to ensure freshness of the attestation report. $\mathcal{P}rv$ executes P on the benign inputs in_b and potentially malicious inputs in_m that are not controlled by $\mathcal{V}rf$ and may lead to the corruption of the program's control flow by means of runtime attacks (§ II). ATRIUM is triggered during the execution of the code region of interest and computes a set of hash measurements over the executed paths. When execution of the code region is complete, $\mathcal{P}rv$ generates and sends to $\mathcal{V}rf$ the final *attestation report* consisting of the concatenated set of hash values $H_0\|...\|H_n$ and the number of iterations of the hash values which correspond to executed loop paths, and a signature over $H_0\|...\|H_n$ and the nonce based on $\mathcal{P}rv$'s secret key sk. To ensure authenticity of the report, sk is stored in memory accessible only by ATRIUM. Upon receiving the report, $\mathcal{V}rf$ verifies its signature using $\mathcal{P}rv$'s public key pk and checks whether the $H_0\|...\|H_n$ values match the reference hash values under input in_b. If they match, $\mathcal{V}rf$ concludes that $\mathcal{P}rv$'s execution of the attested code region was correct in terms of executed instructions and their control flow. For better understanding, we demonstrate next by an example how the hash values are computed during attestation.

Example. A CFG of an example pseudo-code is shown in Figure 3. Each numbered node in the CFG represents the corresponding numbered *basic block* of sequential instructions in the pseudo-code and the address of the first instruction of that basic block. For example, $\mathbf{N}_5$ corresponds to the first 3

[2]https://github.com/xoreaxeaxeax/REpsych
[3]https://github.com/control-flow-attestation/c-flat

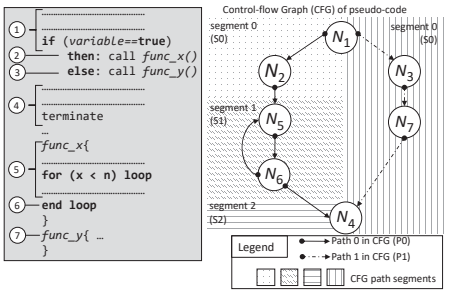


Figure 3: Example pseudo-code and its segmented CFG

instructions outlined in the pseudo-code, constituting a single basic block, and the address of the first instruction. The CFG shown in Figure 3 has 2 main paths: **P0**, in bold, consisting of nodes N_1-N_2-N_5-N_6-N_4 and **P1**, in dashed, consisting of nodes N_1-N_3-N_7-N_4. In order to avoid combinatorial explosion of legal hash values that would occur due to multiple loop iterations, the program CFG is split into segments such that hash values for loop paths are computed separately, rather than computing a single hash value over the complete executed path of the attested region. In Figure 3, due to the loop in N_5-N_6, **P0** is sectioned into 3 segments: $S0$, $S1$ and $S2$. $S0$ comprises all nodes till loop entry at N_5, where $S1$ is initialized. $S1$ ends at the loop exit node N_6, and $S2$ is initialized at N_4 and beyond until again another loop is encountered and so on.

When path **P0** is executed and attested, ATRIUM accumulates nodes (address of the first instruction and the individual instructions in each node) along each segment and computes a hash value for each segment: a hash value $H_0 = H(N_1||N_2)$ over the nodes in $S0$ of **P0**, followed by $H_1 = H(N_5||N_6)$ over the nodes in $S1$, and $H_2 = H(N_4)$ over the nodes in $S2$, resulting in the set of hash values $H_0||H_1||H_2$ representing the executed path **P0**. **P1**, on the other hand, has no loops. Therefore, when executed the whole path is measured by a single hash value $H_3 = H(N_1||N_3||N_7||N_4)$. This CFG segmentation in hash computation allows our scheme to tackle loops and nested loops efficiently, while also allowing fine-grained attestation of their execution. It requires that ATRIUM can detect and interpret loops accurately at runtime. Unlike C-FLAT, we aim to realize this without instrumentation, hence avoiding the associated performance overheads. We present next the architecture of ATRIUM and how it interfaces directly with the processor hardware to capture at runtime every executed instruction and accurately interpret control flow and infer loop entry and exit points *without instrumentation*.

V. ATRIUM: Design and Implementation

ATRIUM is a hardware-based scheme for runtime attestation that tightly integrates with a processor, as shown in Figure 4. This allows it to extract the executed instructions and their memory addresses from the execute stage of the pipeline at runtime while the program P (that needs to be attested) executes on input values in_b and in_m. ATRIUM outputs a set of hash values $H_0\|...\|H_n$ computed over the executed path

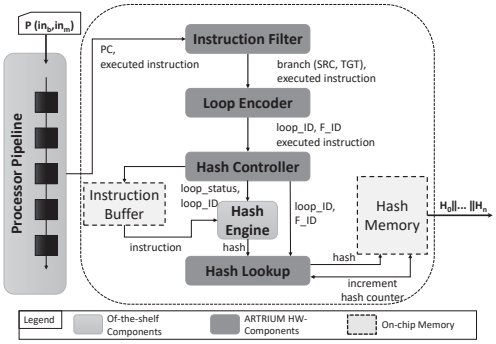


Figure 4: Architecture of ATRIUM

which get included in the attestation report. We present next the components of ATRIUM and their implementation details.

A. Instruction Filter

Upon code execution, the instruction filter extracts the current program counter (PC) and the executed instruction per clock cycle and checks whether the current instruction is a branch or jump, since such instructions reflect control-flow transitions.

Implementation. We implemented the instruction filter such that it tightly extends the execute stage of the processor from which it extracts the PC and instruction per clock cycle. If the current instruction is a control-flow instruction, its PC and the address it jumps to are stored as source–target pair, (Src, Tgt)-pair. To determine whether the branch was taken and whether control jumped forwards or backwards in memory, the PC of the next executed instruction is compared to the stored target address. Instruction filter outputs the following signals: (1) branch instructions, their type, and (Src, Tgt)-pairs and (2) basic block addresses and executed instructions.

B. Loop Encoder

As explained in § IV-B, ATRIUM handles loops and their hash computations differently. Hence, at runtime the loop encoder detects loops and identifies their entry and exit points and their depth, in case of nested loops. It checks whether the behavior of a captured branch can be inferred as returning to a loop's entry point, hence indicating a new loop iteration. The loop encoder instructs the hash controller to finalize the ongoing hash computation and initialize a new one with the entry address of a loop iteration. Furthermore, the loop encoder also detects if a branch represents a system call since system functions have to be handled specially in ATRIUM.

Implementation. To detect loops at runtime without relying on code instrumentation, we utilize a feature of RISC architectures that implement a *link-register*, such as PowerPC, ARM, SPARC, and RISC-V. We adopt a heuristic used in [7] to distinguish between backward branches that indicate loop entry, and branches for subroutine calls where the call target resides earlier in memory. Subroutine calls use instructions that update the *link-register* with the return address, hence, we consider any *non-linking* backwards branch as a *loop entry node*. Consequently, the basic block after the branch instruction is considered a *loop exit node*. This is based on observations

of the RISC-V compiler assembly and its calling convention: any subroutine call with multiple call sites must be *linking* and updates the *link-register*. Subroutines with a single call site can be compiled as a *linking* branch or inlined using the RISC-V compiler. A system call is identified by comparing its target against a predefined list of addresses of such functions and issuing a unique identifier for that function F_ID. The loop encoder stores the addresses of entry and exit nodes of each loop in a content-addressable memory (CAM) to ensure single-cycle constant-access search time. At runtime, every (Src, Tgt) is used to index the CAM to detect if a loop is re-entered or exits and to extract its loop_ID and depth (in case of nested loops). An iterations counter for each loop is maintained and updated at runtime. We detect loop exit when execution proceeds past the currently active loop exit node, either due to sequential execution or a non-linking jump, such as a *break*. The F_ID, loop_ID and loop_status signals are forwarded then to the hash controller.

C. Hash Engine and Hash Controller

The hash engine computes a hash value of each executed path within a segment (§ IV-B). The hash controller regulates the operation of the hash engine, i.e., finalizes or initiates a hash computation based on the control signals received from the loop encoder. In case the computed hash corresponds to a loop path, the hash controller sends this hash to the hash lookup and sets the search boundaries of the hash lookup to that particular current loop (necessary in case of nested loops). Otherwise, the hash value is simply stored in hash memory.

Implementation. We selected Blake2 [4] for hash computations and used the open-source hardware implementation of Blake2b, which takes as an input a message block of size 1Kbit and has a configurable digest size. We configured its digest size to 88 bits to reduce memory requirements for hash lookup and hash memory. The hash controller buffers incoming instructions from the loop encoder, aligns them in 1Kbit message blocks and feeds them to the hash engine. The hash engine requires 28 cycles to process a block, thus the hash controller issues a stall signal to the processor in case its buffer is full and the hash engine cannot digest a new message block. Therefore, system calls are handled differently because we observe that they often involve short loops that are executed arbitrarily many times, e.g., string utility functions. Hashing such a short loop path every time it executes, especially for a large number of iterations, would require the hash controller to stall the processor frequently and delibitate performance. Hence, the executed instructions along a loop path are concatenated and stored in plaintext in a dedicated CAM and sent to the hash engine only once when it is first encountered. When the same path is executed again, it is compared with the previously recorded paths in the CAM, and a corresponding counter is incremented when a match is found, without sending it to the hash engine again. The counters are concatenated with the corresponding hash values in the final attestation report. Upon finalizing a hash computation, the hash controller checks, whether the resulting hash is computed over a path within a loop or not. If it is computed over a path loop, it forwards the resulting hash value from the hash engine synchronized with its corresponding loop_ID to the hash lookup.

[4] https://blake2.net/

D. Hash Lookup

The hash lookup is dedicated to storing and tracking hash values during loop iterations efficiently. Once a hash value is ready, the hash controller forwards it to the hash lookup, which searches within the current loop's list of hash values for a match. If not found, then the hash value is appended to the list. The hash lookup also maintains a counter per loop path which is incremented when its corresponding hash is encountered.

Implementation. To avoid multiple memory accesses due to sequential search of a particular hash value, we implement the hash lookup as a set of CAMs, whose number can be configured based on the system's requirements. A CAM is dedicated for every active loop, so the number of CAMs is determined by the maximum number of nested loops that ATRIUM is configured to track concurrently. Each CAM has a configurable capacity of (n,m) bits, where n is the maximum number of entries and m is number of bits per entry and a counter to maintain the occupied number of entries. When a loop is detected, the hash controller sends the hash lookup to reserve a CAM for it and reset its counter to zero. The CAM holds the computed hash values of a currently executing loop temporarily till the loop exits. Each time a path in the pertinent loop is executed, its computed hash value is looked up in the associated CAM. If a match is not found, i.e., this path has not been executed before, then its hash value is appended to the CAM. When a new loop is detected and all CAMs are occupied, a CAM that was reserved for a loop that already exit (and will not be executed again) is freed and re-used. If a path does not belong to a loop, then its hash value is used to update the hash memory directly.

E. Hash Memory

All computed hash values are stored in a dedicated memory. After the execution of the code region to be attested completes, these hash values are assembled and a digital signature is computed over them. The hash values $H_0\|...\|H_n$ and their signature are then transmitted to $\mathcal{V}rf$.

Implementation. An on-chip hash memory is dedicated to store all computed hash values during a single attestation run of the pertinent code region. The sequence of the storage of the hash values in memory indicates the order of the first occurrence of their corresponding code segments during execution. It is necessary to maintain this order and report $H_0\|...\|H_n$ in the same sequence to $\mathcal{V}rf$ for correctly verifying execution. In our FPGA prototyping of ATRIUM (cf. § VI), we configure the hash memory as on-chip block RAM (BRAM) of configurable capacity with each entry occupying 88 bits for hash digest and 8 bits for its counter. The capacity is configured according to our attestation requirements, i.e., the maximum number of CFG segments an attested code region would consist of. Alternatively, for constrained embedded systems, we can reduce

the memory requirements by streaming the hash values (or every batch of them) as soon as they get generated to the $\mathcal{V}rf$.

VI. Evaluation & Security Considerations

A. Performance & Area Evaluation

We implemented ATRIUM in Verilog, interfaced it with the open-source RISC-V Pulpino core [5], and simulated and synthesized it. Performance and functionality were evaluated using a suite of microprocessor benchmarks including *Dhrystone*, *mt-matmul*, *rsort*, *spvm* and *towers*.

Functionality. We extended the Pulpino RTL with ATRIUM and performed cycle-accurate simulation on ModelSim while executing the aforementioned benchmarks. We confirm correct functionality of ATRIUM by comparing simulation results with reference execution profiles of the benchmarks, which we extracted by running the benchmarks on standalone Pulpino without ATRIUM and analyzing the execution trace.

Area and Memory. Area utilization depends on the configurations of the hash lookup and hash memory of ATRIUM. For our evaluation, we configured the hash lookup with 8 CAMs, each CAM with $n = 8$ entries and each entry being $m = 88$ bits. This allows ATRIUM to track up to 8 active nested loops at once with a maximum of 8 different $88 - bit$ path hashes per loop. On synthesizing ATRIUM using Xilinx Vivado on a Zedboard (Virtex-7 XC7Z020 FPGA), we show the overall area utilization to be 15% of slice registers and 20% of slice LUTs of this FPGA, while only one 18Kbit BRAM is required for the hash memory.

Performance. Implementation results indicate that ATRIUM can operate at a maximum clock frequency of 70 MHz on a Zedboard (Virtex-7 xc7z020 FPGA) and is, hence, on par with the Pulpino's maximum clock frequency of 50 MHz on the same board. Performance experiments show an overhead of 1.97% for *Dhrystone*, 12.23% for *mt-matmul*, 22.69% for *rsort*, 6.06% for *spvm* and 1.7% for *towers*. Since ATRIUM components run on par with Pulpino, performance loss is caused by the hash function, as the processor stalls occur *only* when the currently executed path has ended and needs to be hashed while the hash engine is still processing the previously executed path and is not ready for input. This overhead is incurred for loops with paths whose number of executed instructions are less than the required number of cycles for the hash engine to finalize its computation (28 cycles for the chosen hash function). To mitigate this overhead, the hash engine should be clocked at a higher frequency than the processor if possible.

B. Security Considerations.

We assume that the used cryptographic primitives are secure. Upon receiving an attestation request, $\mathcal{P}rv$ generates and sends the list of computed hash values $H_0 \| ... \| H_n$ along with a digital signature computed over it and a nonce provided by $\mathcal{V}rf$ and signed by $\mathcal{P}rv$'s secret key sk. The signature guarantees the authenticity of the attestation report while the nonce ensures its freshness. By verifying the signature, checking the value of the nonce, and comparing the received hashes to their expected values stored in $\mathcal{V}rf$'s database, $\mathcal{V}rf$ gains assurance of the correct execution (both instruction and their control flow) of the current program on $\mathcal{P}rv$. We consider three classes of attacks that can be mounted on ATRIUM.

Malware and Network Attacks. ATRIUM detects malicious software modification introduced by the adversary, as every executed instruction is included in the hash computation. To evade detection, finding a second image that maps to same hash value is required. However, that is infeasible since the hash engine is second pre-image resistant. Forging the signature or replaying an old signature is also not feasible, due to security of signature scheme and to the nonce being long enough.

Runtime Attacks. Since basic block addresses are included in hash computations along with the executed instructions, the hash values computed in ATRIUM reflect the control flow of the executed path. Being tightly integrated with the processor, ATRIUM is guaranteed to track and record every control-flow event executed. An attacker who knows the program code P or CFG(P) can try to bypass ATRIUM by searching for a second pre-image of the corresponding hash. However, by using cryptographically-secure hash function, finding collisions is computationally infeasible.

Physical Attacks. An adversary with physical access to $\mathcal{P}rv$ can try to manipulate the program code in $\mathcal{P}rv$'s memory at runtime, i.e, between time of attestation and time of execution. However, in ATRIUM attestation is performed *during* execution. Therefore, it is guaranteed that *every instruction* that is executed on $\mathcal{P}rv$ will be included in the hash generation, and consequently any manipulation will be detected by $\mathcal{V}rf$, as the generated hash values would not match $\mathcal{V}rf$'s expectations. This defends against TOCTOU attacks that can occur when attestation is *followed* by execution, as was the case for both SMART [9] and C-FLAT [3]. Finally, fault injection attacks which target the SoC clock and cause unintended behavior would also be detected by $\mathcal{V}rf$, as long as the attacks affect the instructions executed or their control flow. Note that, expensive invasive/semi-invasive physical attacks on the SoC are considered out of scope in this work.

VII. Related Work

Attestation Schemes. Existing static attestation schemes such as software-based [14], [20], hardware-based [21], [17], and hybrid [15], [9] attestation schemes are vulnerable to runtime attacks. Control-flow attestation (C-FLAT) aims at enhancing the security of static attestation schemes by additionally hashing the code's execution control flow. This enables the detection of code-reuse and non-control data attacks that divert the execution flow. However, due to frequent hash calculations and context switching (on TrustZone), C-FLAT incurs high performance overhead. LO-FAT [7] leverages hardware assistance to track and measure control flow, thus, overcoming the limitations of C-FLAT and enabling efficient attestation of *uninstrumented* code. LO-FAT, however, incurs significant area overhead due to its on-chip memory requirements (up to 49 36Kbit Block RAMs are used sparsely to store counters of

[5] https://github.com/pulp-platform/pulpino

loops' paths). Finally, in a stronger adversary model with physical access to the prover's device, these schemes are vulnerable to Time of Check Time of Use (TOCTOU) attacks. ATRIUM mitigates this by providing both static and control-flow attestation in a stronger (and more realistic) adversary model efficiently.

Authenticated Memory Modules. Authenticated Memory Modules (such as Intel Authenticated Flash [1]) aim at resisting physical attacks on external memory by preserving the memory's integrity. However, they are insecure under an adversary model with physical access. Moreover, they do not authenticate the control flow of the execution. On the contrary, ATRIUM provides an additional defense against software runtime attacks by coupling the attestation of both the instructions and their control flow with their execution to eliminate any room for TOCTOU attacks.

Memory Authentication. Such schemes [8], [6] aim at resisting physical attacks on external memory. However, they incur high performance overhead by authenticating memory blocks before execution and are susceptible to runtime attacks. ATRIUM detects both runtime attacks and physical attacks on code memory while incurring minimal overhead.

Hardware Security Architectures. Finally, hardware security architectures (such as Intel SGX) provide memory authentication as well as static attestation. However, such architectures are not designed to target low-end embedded devices. Furthermore, they only provide static attestation and therefore cannot meet the goals that we target. Nevertheless, they provide security features complementary to our work.

VIII. Conclusion

Due to the ubiquity of interconnected embedded systems, software running on these devices have become vulnerable to remote software attacks. Previous attestation schemes have been proposed to detect these attacks while always ruling out physical attacks. In this paper, we showed that physical attacks on the system's code memory are indeed feasible. We presented a hardware-based efficient scheme ATRIUM that allows precise attestation of both executed instructions as well as their control flow. ATRIUM is the first attestation scheme to provide security guarantees against a stronger adversary with physical access to code memory, and does not require any code instrumentation (compliant to legacy software) or instruction set extension. Our proof-of-concept implementation is highly efficient with reasonable performance impact on the attested software at an expense of minimal area overhead and memory.

Acknowledgments. We thank the authors of SMART for sharing their scheme. This work was supported by the German Science Foundation CRC 1119 CROSSING, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant No. 643964 (SUPERCLOUD), the U.S. Department of Energy through the Early Career Award (DE-SC0016180), the National Science Foundation Graduate Research Fellowship Program under grant No. 1144246, and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

References

[1] Intel Authenticated Flash. www.design-reuse.com/articles/16975.
[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "Control-flow integrity: Principles, implementations, and applications". *ACM Transactions on Information and System Security*, 2009.
[3] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. "C-FLAT: Control-flow attestation for embedded systems software". In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.
[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. "When good instructions go bad: Generalizing return-oriented programming to RISC". In *ACM SIGSAC Conference on Computer and Communications Security*, 2008.
[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. "Non-control-data attacks are realistic threats". In *USENIX Security Symposium*, 2005.
[6] R. de Clercq, R. De Keulenaer, P. Maena, B. Preneel, B. De Sutter, and I. Verbauwhede. "SCM: Secure code memory architecture". In *ACM Symposium on Information, Computer and Communications Security*. ACM, 2017.
[7] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi. "LO-FAT: Low-overhead control flow attestation in hardware". In *Design Automation Conference*, 2017.
[8] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres. "Hardware mechanisms for memory authentication: A survey of existing techniques and engines". In *Transactions on Computational Science IV*. 2009.
[9] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. "SMART: Secure and minimal architecture for (establishing dynamic) root of trust". In *Annual Network and Distributed System Security Symposium*, 2012.
[10] A. Francillon and C. Castelluccia. "Code injection attacks on Harvard-architecture devices". In *ACM SIGSAC Conference on Computer and Communications Security*, 2008.
[11] V. Haldar, D. Chandra, and M. Franz. "Semantic remote attestation: A virtual machine directed approach to trusted computing". In *Virtual Machine Research And Technology Symposium*, 2004.
[12] Hewlett-Packard. "Data execution prevention", 2006.
[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. "Data-oriented programming: On the effectiveness of non-control data attacks". In *IEEE Symposium on Security and Privacy*, 2016.
[14] R. Kennell and L. H. Jamieson. "Establishing the genuinity of remote computer systems". In *USENIX Security Symposium*, 2003.
[15] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. "TrustLite: A security architecture for tiny embedded devices". In *ACM SIGOPS EuroSys*, 2014.
[16] T. Kornau. "Return oriented programming for the ARM architecture". Master's thesis, Ruhr-University Bochum, 2009.
[17] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. "New results for timing-based attestation". In *IEEE Symposium on Security and Privacy*, 2012.
[18] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. "Code-pointer integrity". In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.
[19] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. "SoK: Automated software diversity". In *IEEE Symposium on Security and Privacy*, 2014.
[20] Y. Li, J. M. McCune, and A. Perrig. "VIPER: Verifying the integrity of peripherals' firmware". In *ACM SIGSAC Conference on Computer and Communications Security*, 2011.
[21] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. "Copilot – A coprocessor-based kernel runtime integrity monitor". In *USENIX Security Symposium*, 2004.
[22] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. "Design and implementation of a tcg-based integrity measurement architecture". In *USENIX Security Symposium*, 2004.
[23] H. Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In *ACM SIGSAC Conference on Computer and Communications Security*, 2007.
[24] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal war in memory". In *IEEE Symposium on Security and Privacy*, 2013.

C

LITEHAX: LIGHTWEIGHT HARDWARE-ASSISTED ATTESTATION OF PROGRAM EXECUTION

[38] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution. In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2018. Core Rank A.

LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution

Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, Ahmad-Reza Sadeghi

Technische Universität Darmstadt, Germany
{ghada.dessouky, tigist.abera, ahmad.ibrahim, ahmad.sadeghi}@trust.tu-darmstadt.de

ABSTRACT

Unlike traditional processors, embedded Internet of Things (IoT) devices lack resources to incorporate protection against modern sophisticated attacks resulting in critical consequences. Remote attestation (RA) is a security service to establish trust in the integrity of a remote device. While conventional RA is static and limited to detecting malicious modification to software binaries at load-time, recent research has made progress towards runtime attestation, such as attesting the control flow of an executing program. However, existing control-flow attestation schemes are inefficient and vulnerable to sophisticated data-oriented programming (DOP) attacks subvert these schemes and keep the control flow of the code intact.

In this paper, we present LiteHAX, an efficient hardware-assisted remote attestation scheme for RISC-based embedded devices that enables detecting both control-flow attacks as well as DOP attacks. LiteHAX continuously tracks both the control-flow and data-flow events of a program executing on a remote device and reports them to a trusted verifying party. We implemented and evaluated LiteHAX on a RISC-V System-on-Chip (SoC) and show that it has minimal performance and area overhead.

1 Introduction

The proliferating rise of the Internet of Things (IoT) hype has made embedded devices increasingly ubiquitous and deployed in numerous settings. These devices collect, process, and communicate security, privacy and safety critical information and due to their pervasiveness, connectivity, and increased sensing and actuating capabilities, they provide an attractive attack surface. On the other hand, to meet the cost and power consumption budgets, embedded devices are usually resource-constrained and lack sophisticated security features of legacy computing devices. This has made embedded device security particularly challenging in the face of various known and emerging attack strategies, e.g., malware infestation, control-flow hijacking, and data-oriented programming (DOP) attacks [13]. Critical exploits include Stuxnet[1] and the more recent *Mirai* malware[2] in 2016, where a series of disruptive Distributed Denial-of-Service (DDoS) attacks were committed, by compromised IoT devices, including routers and web-enabled security cameras, against the DNS system. Successor variants of Mirai such as *Satori*[3] and *Okiru*[4] have been crafted that target popular embedded processors from ARM to x86 among others to exploit many more IoT devices.

Traditionally runtime attacks exploit a security vulnerability, typically memory corruption, and modify the code on a device by injecting malicious code. However, since deployed protection mechanisms such as Data Execution Prevention (DEP) [16] have proven effective against code-injection attacks, attackers have resorted to other tactics such as code-reuse techniques like Return-Oriented Programming (ROP) [28]. These techniques exploit vulnerabilities to corrupt control-data and re-use the code chunks already residing in the memory of the targeted program to build the attack payload and hijack the control flow of the program. More recently, practical Data-Oriented Programming (DOP) attacks were shown [13] which allow the adversary to execute Turing-complete malicious execution by carefully corrupting only non-control-data to stitch a sequences of operations on attacker-controlled input. DOP attacks do not divert the program's control flow or modify its binaries. To mitigate these sophisticated attacks, there has been ongoing intensive research on runtime attacks and defenses in recent years. Prominent defense approaches are control-flow integrity [1] (CFI), code-pointer integrity [22], and (fine-grained) code randomization [9, 23] to name some. However, these solutions enforce security policies such as control-flow integrity and neither provide any information about the complete state of a program's execution (e.g., required in detecting non-control-data attacks) nor can they mitigate DOP attacks without generating prohibitively high performance overhead [7, 13].

Remote attestation (RA) is a security service that aims at *detecting* malware infestation. It is based on an interactive protocol through which a remote device (the prover) sends an authenticated report about its software configuration (i.e., usually an authenticated hash) to a trusted entity (the verifier) to prove that it has not been altered. Conventional attestation schemes are static and rely on the binary digest of the code at load-time. Recent advances in attestation solutions have aimed to attest the runtime behavior of program execution by reporting the program's control-flow path [2, 10, 33] and detecting control-flow attacks as well as some non-control-data attacks that change the control flow to a valid, yet unauthorized control-flow path. However, this still leaves the mitigation of highly expressive DOP attacks an open problem, while DOP attacks are likely to become the next prevalent attack technique as control-flow defenses become more established.

In this paper, we propose LiteHAX– a hardware-assisted scheme enabling remote runtime attestation on RISC-based embedded devices. LiteHAX allows to securely and efficiently record and report prover's control- and data-flow events to a remote verifier. In con-

[1] http://www.computerworld.com/article/2519574/security0/stuxnet-renews-power-grid-security-concerns.html
[2] https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html

[3] https://www.computerweekly.com/news/450431409/Next-gen-Mirai-botnet-sparks-calls-for-more-secure-IoT-design
[4] https://thehackernews.com/2018/01/mirai-okiru-arc-botnet.html

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

DOI: https://doi.org/10.1145/3240765.3240821

trast to existing schemes [2, 10, 33], this allows a trusted verifier to detect (1) control-data attacks that overwrite a code pointer such as ROP [28], (2) non-control-data attacks that indirectly effect the control flow of a program, and (3) existing data-only attacks such as DOP [13, 14]. The intuition behind our work is that all known and reported non-control-data and DOP attacks are essentially reduced to corrupting *memory access operations*, without inflicting any unintended control flow [13]. In RISC-based architectures, which is the target architecture of the prototype in this work, memory accesses are only possible via load and store instructions.

LiteHAX is minimally invasive, i.e., it does not require modifications to the processor micro-architecture, extensions to the instruction set architecture, or instrumentation of the program code. Furthermore, unlike existing runtime attestation schemes such as C-FLAT [2], LiteHAX is applicable to a more realistic embedded device usage scenario, as it allows the verifier to continuously monitor the code executing on the prover while inducing minimal overhead in terms of runtime, area, and memory requirements on both entities.

Control-flow hijacking attacks are detected because the verifier receives an encoding of the control-flow path executed at the prover which it uses to re-construct and validate the execution path. By performing online symbolic execution and data-flow analysis that is constrained to the re-constructed control-flow path to generate the reference legal memory access operations, LiteHAX overcomes the problem of execution-path explosion induced by control-flow attestation [2] while also providing context-sensitive security guarantees. DOP attacks are detected because the verifier receives a short digest that represents all the memory access operations that have been actually executed by the prover and compares them with the aforementioned generated reference.

The main contributions of our work are:

- *Data-flow attestation:* We present LiteHAX– the first runtime attestation scheme that is capable of detecting runtime attacks that do not change the control flow of the executing program.
- *Proof of concept:* We implement LiteHAX on the Pulpino core – an open-source RISC-V microcontroller-based SoC.
- *Systematic evaluation:* We present an evaluation of LiteHAX in terms of security, performance and hardware overhead. Evaluation results show the efficiency and practicality of our design and implementation.

2 Problem Description

We present next an overview of the different classes of software runtime attacks that we aim to detect and proposed defenses to date.

Control-Hijacking Attacks. Runtime attacks exploit program vulnerabilities to corrupt the memory space and cause malicious behavior. The most popular entry-point to a vulnerable program is via a buffer overflow where the attacker writes data to a buffer on the stack or heap beyond its intended bounds and corrupts adjacent memory locations. These attacks usually aim to manipulate the control-flow information stored on the program's stack and heap in order to hijack the intended control flow of the program execution.

Figure 1 demonstrates the typical memory layout of a C program and the different classes of runtime attacks that it is vulnerable to. Exclusive memory sections are dedicated for the data and code segments of a given program. The former is assigned *read & write* permissions and the latter is assigned *read & execute* permissions. This ensures that code cannot be executed from data memory and that code memory cannot be maliciously overwritten by means of a software adversary. A program can be analyzed by static and dynamic analysis to generate its corresponding control-flow graph (CFG) that dictates the valid control-flow paths a program should follow while executing. The numbered nodes (N_1...N_8) in the CFG

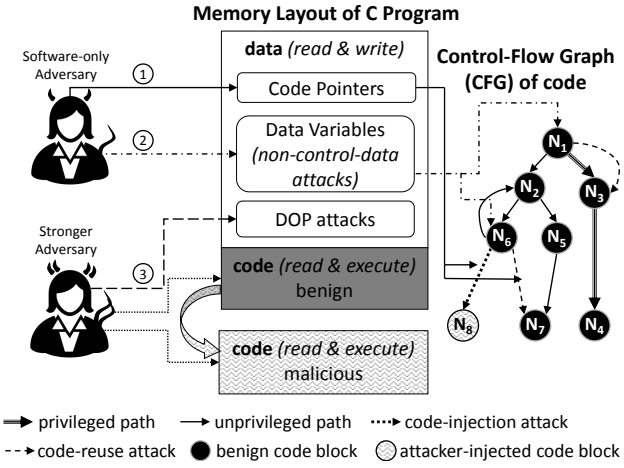


Figure 1: Different classes of runtime attacks

in Figure 1 represent the basic blocks of the code, while the edges represent the control-flow transitions from one block to the next by means of a control-flow instruction. A valid control-flow path is any path that exists within the CFG. However, not all valid paths are necessarily legitimate in a given execution context.

Runtime attacks can be categorized into: ① code-pointer overwrite attacks, ② non-control-data attacks which corrupt data variables to indirectly affect the control flow, and ③ Data-Oriented Programming (DOP) attacks which *do not* affect the control flow.

By corrupting control-flow information stored in the stack or heap and overwriting code-pointers (return addresses or function pointers) as in ①, an adversary can redirect the control flow of a program such that execution has a malicious and unauthorized effect. This is possible in one of two ways; either via *code-injection* attacks or *code-reuse attacks*. In *code-injection attacks*, the adversary injects a malicious executable payload in program memory (node N_8) and redirects control flow to execute it after node N_6, as shown in Figure 1. Alternatively, in state-of-the-art *code-reuse* attacks, such as *Return-oriented Programming* (ROP) [28], the adversary stitches together a malicious sequence of machine code instructions from benign *gadgets* of code already residing in the memory of the vulnerable program. To achieve this, the adversary would redirect control flow to execute the already existing benign code represented by node N_7 after executing node N_6. All of the above attacks result in the control flow explicitly being hijacked and redirected to an invalid path that does not exist in the CFG. Such runtime exploitation attacks have been shown to be a threat on many popular processor architectures, such as Intel x86 [28] and ARM [20] among others.

In response to these control-flow hijacking attacks, various principled defenses have been proposed in recent years. Code-injection attacks are prevented by marking memory as writable or executable using W⊕X memory access policies such as Data Execution Prevention (DEP) [16]. Code-reuse attacks are mitigated by defenses such as *Control-Flow Integrity* (CFI) [1, 18], Address Space Layout Randomization (ASLR) [23], and Code-Pointer Integrity (CPI) [22].

Non-Control-Data Attacks. While the above attacks (and their defenses) focus on the control plane of program's execution, an adversary naturally is compelled to investigate next its capabilities within the data plane of the execution. An adversary could corrupt critical data variables that drive the control flow of the execution via the more sophisticated *non-control-data attacks* [8] as in attacks ②. This may redirect the control flow to a *valid*, yet illegal and unintended path in the given execution context. An adversary may corrupt a critical authentication variable (at node N_1) and redirect execution to continue in a privileged path (nodes $N_1 \rightarrow N_3 \rightarrow N_4$)

even though the user has not been authenticated to execute this path and should have executed the unprivileged path (nodes $N_1 \rightarrow N_2 \rightarrow N_5$) instead. Alternatively, an adversary may also corrupt a loop counter variable (at node N_6) to modify the number of iterations a program loop executes. The attack payloads here are, however, simple and limited to corrupting a critical data variable and causing privilege escalation or sensitive data leakage.

Data-Oriented Programming (DOP) Attacks. In *DOP attacks* as in ③, the adversary carefully corrupts non-control-data to chain sequences of instructions (*data-oriented gadgets*) to execute highly expressive (assignment, arithmetic and conditional branching) operations on some attacker-controlled input. The key challenge is in crafting such an expressive construction of the desired malicious execution without incurring any illegitimate control flow with respect to the CFG. The data-oriented gadgets used consist of sequences of operations which can be visualized as a single virtual machine instruction executing on top of benign program logic. An adversary-controlled loop in the benign program, known as a *gadget dispatcher*, is used to stitch together the data-oriented gadgets and realize expressive computation and malicious behavior. Hu et al. [13] demonstrated three end-to-end DOP exploits against the ProFTPD file server and one DOP attack against the Wireshark network packet analyzer. Evans [14] demonstrated another attack against the GStreamer FLIC decoder.

To date, deterring non-control-data attacks, and even stealthier DOP attacks, remains a significant challenge as they have been shown to actively break state-of-the-art defenses. While runtime attestation schemes, C-FLAT [2] and LO-FAT [10], may detect *some* non-control-data attacks, the more critical DOP attacks cannot be detected by them. With defenses against control-flow hijacking attacks on constrained embedded devices becoming increasingly available, it is only natural for adversaries to turn to crafting DOP attacks on embedded systems.

3 LiteHAX: Our Scheme

The intuition behind our work is that all known and reported non-control-data and DOP attacks only corrupt *memory load and store operations*, without inflicting any unintended control flow. To that end, we observe that it has become insufficient to only attest control flow in runtime attestation schemes. We present LiteHAX, a runtime attestation scheme that continuously captures and attests both control flow and data flow of any given program execution in an efficient and lightweight approach that is well suited for low-end embedded devices. While the control-flow events would explicitly reflect any control-flow hijacking exploits, the fine-grained memory operations' trace would reflect illegal memory accesses that result from memory corruption vulnerabilities being exploited as an entry-point for an adversary to craft a data-oriented or a DOP attack.

LiteHAX is deployed by extending the processor core of a remote in-field embedded device (called the prover $\mathcal{PRV}$) with custom hardware that tracks and records the fine-grained control- and data-flow events of executing programs at runtime. The recorded execution trace is then reported to a trusted third party (called the verifier $\mathcal{VRF}$), which in turn verifies that the reported execution is as expected, i.e., whether the reported control- and data flows are legal for the given execution context. LiteHAX builds on the threat model and assumptions that we describe next.

3.1 Threat Model and Assumptions

We assume that both $\mathcal{VRF}$ and $\mathcal{PRV}$ have access to the source and binary code of the target program and that conventional static (binary) attestation is deployed to assure that $\mathcal{PRV}$ is executing unmodified program `Prog`. We also assume that address space layout randomization (ASLR) is not deployed. This assumption is reasonable since low-end embedded systems targeted by LiteHAX do not currently support ASLR due to their limited processing power. Assuming no ASLR guarantees that $\mathcal{VRF}$ has access to the memory address space mapping of the target program on $\mathcal{PRV}$. Otherwise, addresses would change for each run of the program and instrumentation can be used to embed unique labels to identify basic blocks and memory access operations. We assume $\mathcal{PRV}$ has data execution prevention (DEP) deployed to prevent injecting malicious code into running processes. Finally, for simplicity, we focus on RISC-based load-store architectures for prototyping, where only load and store instructions access data memory.

We consider a powerful adversary $\mathcal{ADV}$ with full control over the *data memory* of the target program executing on $\mathcal{PRV}$. $\mathcal{ADV}$ can launch runtime attacks (§ 2) by exploiting standard memory corruption vulnerabilities (e.g., externally-controlled format string[5]) that cause buffer overflows leading to corruption of data memory. We assume $\mathcal{ADV}$ cannot modify program code at runtime (due to W⊕X protection). Furthermore, $\mathcal{ADV}$ cannot modify hardware-protected memory used exclusively by LiteHAX. This assumption is valid since this memory is not mapped to software-accessible address space and invasive physical attacks are out of scope.

3.2 LiteHAX Attestation Scheme

We derive the following requirements for a secure runtime attestation scheme:

- *Runtime security:* The scheme should be capable of detecting all runtime attacks throughout the program execution, both control-flow and data-oriented attacks. Continuously tracking and recording both control- and data-flow events of execution is sufficient to reveal all such runtime attacks.
- *Accuracy & completeness:* It should accurately record every control- and data-flow event. This is guaranteed by integrating the attestation hardware modules tightly with the processor.
- *Secure reporting:* It should securely report attestation results which are integrity-protected and fresh. This is achieved by using digital signatures and a monotonic counter.
- *Low overhead on prover:* It should incur minimal performance overhead on the low-end $\mathcal{PRV}$ device. This is made possible by leveraging hardware-assisted extensions for tracking and recording control- and data-flow events.
- *No state explosion:* It should not yield an explosion of possible attestation reports stored and checked by the verifier. This is achieved by performing online context-sensitive analysis and symbolic execution constrained to the reported control-flow path.

LiteHAX has two phases: an offline phase, where $\mathcal{VRF}$ generates the necessary information to verify attestation reports (control-flow graph – CFG); and an online phase, during which $\mathcal{PRV}$ sends its execution trace (i.e., control flow and data flow) to be validated by $\mathcal{VRF}$. The offline phase is executed only once by $\mathcal{VRF}$, and the online phase is executed continuously between the $\mathcal{VRF}$ and $\mathcal{PRV}$. We illustrate the LiteHAX attestation protocol in Figure 2.

Offline Phase. The verifier $\mathcal{VRF}$ performs a one-time preprocessing to generate the CFG of the target program `Prog` by means of static and dynamic analysis. We do *not* require $\mathcal{VRF}$ to generate a data-flow graph (DFG) or create a database of the cryptographic hash measurements over all control- and data-flow events. This would result in a combinatorial explosion of the number of valid hash measurements, even if the different execution paths for each loop were only considered (excluding their order of execution

[5] CWE-134: Use of Externally-Controlled Format String https://cwe.mitre.org/data/definitions/134.html

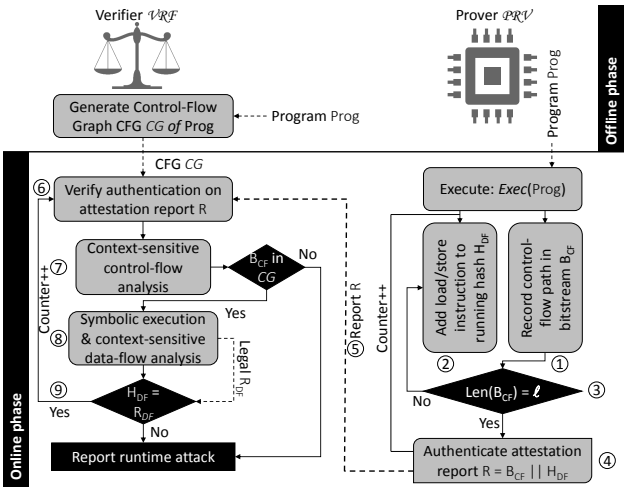


Figure 2: Attestation protocol of LiteHAX

and iteration counts) [2]. Alternately, $\mathcal{VRF}$ only generates the legal sequence of memory load/store instructions in the *online* phase by performing symbolic execution and data-flow analysis constrained along the reported execution path (see online phase below).

Online Phase (Prover-side). During execution, LiteHAX tracks and records every control-flow transition executed by $\mathcal{PRV}$ and encodes it in the form of a bitstream $\mathtt{B_{CF}}$ in a dedicated memory buffer (operation ① Figure 2). Simultaneously, LiteHAX creates a hash measurement $\mathtt{H_{DF}} = \mathtt{hash}(\mathtt{Load}_1/\mathtt{Store}_1 \| \ldots \| \mathtt{Load}_n/\mathtt{Store}_n)$ that is computed over all the executed load/store instructions (operation ②). The recorded bitstreams $\mathtt{B_{CF}}$ allow $\mathcal{VRF}$ to re-construct the exact execution path executed at $\mathcal{PRV}$ segment by segment and perform context-sensitive analysis at runtime. Because $\mathcal{VRF}$ keeps state of the execution context and control flow throughout, it can verify that each control transfer is benign and consistent with the program's executed control-flow path up until that point in time, i.e., that it adheres to a context-sensitive CFG. On the other hand, $\mathtt{H_{DF}}$ allows $\mathcal{VRF}$ to verify the state of the data flow on $\mathcal{PRV}$.

$\mathcal{PRV}$ regularly (e.g., at specific time periods or when $\mathtt{B_{CF}}$ reaches a configured length ℓ ③) sends to $\mathcal{VRF}$ an intermediate *attestation report* $\mathtt{R} = \{\mathtt{B_{CF}}, \mathtt{H_{DF}}, \sigma\}$ formed of the control-flow bitstream $\mathtt{B_{CF}}$ and the data-flow hash measurement $\mathtt{H_{DF}}$ (⑤). The report is authenticated along with the monotonic counter `ctr` by $\mathcal{PRV}$ using a digital signature σ based on its signing key $sk_{\mathcal{PRV}}$ (④). To ensure authenticity of the report, $sk_{\mathcal{PRV}}$ is stored in hardware-protected memory that is only accessible by LiteHAX.

Online Phase (Verifier-side). Upon receiving the report, $\mathcal{VRF}$ verifies the signature σ using $\mathcal{PRV}$'s public key $pk_{\mathcal{PRV}}$ (⑥) for authenticity. $\mathcal{VRF}$ then re-constructs the executed path encoded in $\mathtt{B_{CF}}$ by stitching it to the last control-flow execution state. It utilizes a runtime analysis to refine the CFG with context-sensitive analysis and dynamically activates executed control-flow transfers and basic blocks on the CFG while validating them. This constrains all online analysis only to the activated segment of the CFG which eliminates the exponential increase in analysis time and state explosion that would otherwise occur, while still enabling $\mathcal{VRF}$ to perform context-sensitive control-flow validation, e.g., call-return matching (⑦). Simultaneously, $\mathcal{VRF}$ runs symbolic execution and incremental forward data-flow analysis constrained by the execution control flow encoded in $\mathtt{B_{CF}}$. Constrained symbolic execution enables $\mathcal{VRF}$ to generate the memory addresses that were accessed by each executed instruction during the actual execution. The data-flow analysis generates the data objects that each instruction is allowed to access. Since field-insensitive and array-insensitive data-flow analysis cannot distinguish between fields of an object or elements of an array, $\mathcal{VRF}$ then checks whether the memory addresses obtained from the symbolic execution are within the memory bounds allocated for the allowed object (⑧). If not, this would indicate a data-oriented attack. $\mathcal{VRF}$ computes a reference hash measurement $\mathtt{R_{DF}}$ over the instructions, their addresses, and the data addresses they access. Finally, $\mathcal{VRF}$ checks whether the received hash measurement $\mathtt{H_{DF}}$ matches $\mathtt{R_{DF}}$ (⑨). A mismatch would indicate a runtime attack.

4 LiteHAX: High-Level Architecture

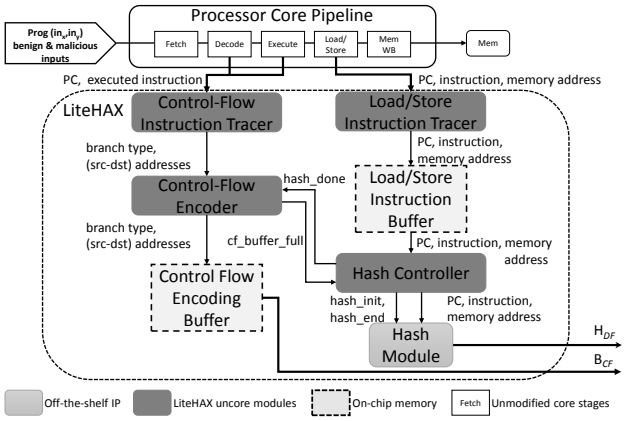


Figure 3: Architecture of LiteHAX

Figure 3 shows our hardware architecture for LiteHAX at $\mathcal{PRV}$'s end. LiteHAX works by continuously tracking and recording control-flow and data-flow instructions at runtime while program `Prog` executes on its benign inputs and potentially malicious adversary-controlled inputs.

LiteHAX takes advantage of the control-flow and load/store tracking features inherent in any pipelined processor and interfaces with the decode, execute and load/store stages, as shown in Figure 3. It extracts and encodes control-flow events at runtime in a bitstream $\mathtt{B_{CF}}$. Simultaneously, it tracks all memory access events (which represent data flow) and records them compactly by computing a cumulative cryptographic hash measurement $\mathtt{H_{DF}}$ over them. $\mathcal{PRV}$ then sends to $\mathcal{VRF}$ an *attestation report* $\mathtt{R} = \{\mathtt{B_{CF}}, \mathtt{H_{DF}}, \sigma\}$ (cf. § 3.2).

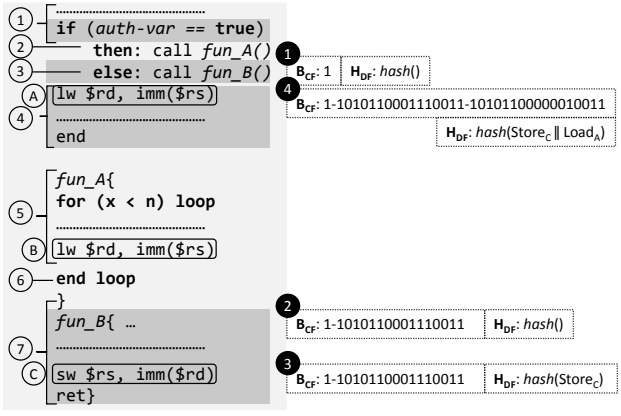


Figure 4: Example pseudo-code and its generation of $\mathtt{B_{CF}}$and $\mathtt{H_{DF}}$

The hardware add-on required is primarily comprised of uncore modules that are tightly integrated with the processor core while being minimally invasive to the core itself, i.e., requiring no modifications to the processor pipeline or instruction set architecture extensions. The key modules are the *control-flow instruction tracer*,

control-flow encoder, *load/store instruction tracer* and off-the-shelf hardware cryptographic *hash module* as shown in Figure 3.

We describe the high-level operation of LiteHAX and how it attests the execution of an example pseudo-code shown in Figure 4. Each basic block of instructions is denoted by a numbered node and each memory access instruction is denoted by a lettered node. During execution, the control-flow instruction tracer extracts the current program counter, the executed instruction and its type at every clock cycle from the decode and execute stages. Then it filters in and captures only control-flow instructions, namely all `branch`, direct and indirect `jump` and `return` instructions, and records their type and source `SRC` and destination `DST` addresses and forwards this to the control-flow encoder.

The control-flow encoder efficiently encodes control-flow events into a bitstream that is compact yet sufficient to accurately reconstruct the control-flow path executed. Both conditional branches and direct jumps are encoded with a single bit. If the conditional branch is not taken and execution proceeds sequentially, this transfer is encoded as a '0'. If the conditional branch is taken and execution proceeds non-sequentially, this is encoded as a '1'. Note that, all direct jumps must jump to their hard-coded addresses and are, therefore, always encoded as a '1'. However, we cannot eliminate the encoding of direct jumps altogether because this might result in ambiguous re-construction at $\mathcal{VRF}$ if the last control-flow transfers executed (for a transferred bitstream) were non-encoded direct jumps. Indirect jumps require that the destination address is encoded in 32-bit, since the `jump and link register (JALR)` instruction in the RISC-V ISA is defined to enable a two-instruction sequence of `load upper immediate (LUI)` followed by a `JALR` to jump anywhere in a 32-bit absolute address range. This encoding guarantees that the minimum data required for an accurate re-construction of the executed control-flow path is stored and streamed to $\mathcal{VRF}$.

After the conditional branch instruction `if (auth-var == true)` in basic block N_1 in Figure 4 is evaluated, control flow either proceeds to node N_2 or N_3. If `auth-var` is evaluated to be false, then execution jumps to the base address of N_3. Since the conditional branch jumps non-sequentially to N_3, this gets encoded as a '1' into a dedicated $\mathtt{B_{CF}}$ memory buffer. At N_3 `fun_B()` is called and control flow is redirected to the base address of node N_7. This indirect jump is encoded by its `DST` address (base address of N_7) "1010110001110011" which gets appended to the $\mathtt{B_{CF}}$ bitstream. The buffer continues to accumulate the bits generated for every control-flow transfer and gets read out later when the attestation report is being assembled.

Simultaneously, the load/store instruction tracer extracts from the load/store stage every executed load/store instruction, its address in code memory, and the data memory address it accessed, and buffers this information into a dedicated memory. If we were to naively store and transfer this information as is $\mathcal{PRV}$ would be streaming very large attestation reports to $\mathcal{VRF}$. Instead, to generate a compact attestation report that can be efficiently streamed and verified, we employ a cryptographic hash module to compute a cumulative hash measurement $\mathtt{H_{DF}}$ over these data-flow events.

In Figure 4, while executing `fun_B()` in N_7, the memory access instruction Ⓒ `sw $rs, imm($rd)` is captured by the load-/store instruction tracer. It gets stored in the load/store instruction buffer followed by subsequent data-flow events (such as Ⓐ next in N_4) which continue to get absorbed into the hash module until the control-flow encoding buffer is full. This is indicated to the hash controller by the `cf_buffer_full` signal. The hash controller regulates the operation of the hash module and synchronizes it with other modules of LiteHAX. When the hash controller receives the signal `cf_buffer_full` asserted, it communicates to the hash module to finalize (via `hash_end` signal) the running hash computation, i.e., no more data-flow events are absorbed in generating this particular hash measurement. Incoming data-flow events in the meantime are buffered into the dedicated load/store instruction memory and are forwarded to the hash module only when it is ready again. When the hash computation is completed and the data-flow hash measurement $\mathtt{H_{DF}}$ is successfully generated and read out, this is indicated by the hash controller to the control-flow encoder via `hash_done` signal. Consequently, the corresponding control-flow bitstream $\mathtt{B_{CF}}$ is also read out as shown in Figure 3.

The execution is effectively traced and measured in segments where a set $\mathtt{B_{CF}}$ and $\mathtt{H_{DF}}$ measurements are continuously generated for subsequent execution segments and transferred to $\mathcal{VRF}$. Besides enabling continuous attestation of the executed program, LiteHAX also tackles the challenges associated with fine-grained attestation of loops and recursion because control flow is recorded and reconstructed at $\mathcal{VRF}$ for as many times as a loop iterates.

We present next our PoC implementation of LiteHAX.

5 LiteHAX: Implementation

5.1 Prover Implementation

Control-Flow Instruction Tracer. The control-flow instruction tracer is implemented such that it directly interfaces with the decode and execute stages of the processor core to collect information on the type and operands of each instruction from the decode stage. If the current instruction is a control-flow instruction, the program counter (PC), and the address it jumps to in the next branch execution cycle are stored in a 64-bit register as a source–destination pair – `SRC`, `DST`. If the `SRC`, `DST` addresses are consecutive this indicates that the branch was not taken and non-consecutive `SRC`, `DST` addresses indicate that the branch was taken. In the unlikely case that speculative execution is supported on an embedded core, extracting `DST` address is delayed by a pre-defined number of clock cycles until execution is committed.

Control-Flow Encoding and Buffering. We described earlier in § 4 how different control-flow instructions are encoded into the bitstream $\mathtt{B_{CF}}$. To prevent dropping of any control-flow events, the $\mathtt{B_{CF}}$is buffered into a dedicated memory as it gets generated. In our FPGA prototype (cf. § 6), we implement this as a First-In-First-Out (FIFO) buffer using on-chip block RAM (BRAM) of configurable capacity (cf. § 6). When the buffer is full, $\mathtt{B_{CF}}$ becomes ready to be read out and streamed to $\mathcal{VRF}$. However, the corresponding data-flow hash computation $\mathtt{H_{DF}}$ must be finalized first before $\mathtt{B_{CF}}$ can be read out to prepare the attestation response. In the meantime, LiteHAX must continue to attest the execution without dropping any control-flow (or data-flow) events, thus, the control-flow encoding buffer should remain available to store the continuously generated bitstream. We guarantee this by configuring the capacity of the control-flow encoding buffer such that it exceeds the configured length of $\mathtt{B_{CF}}$ to compensate for the maximum number of clock cycles incurred by the hash module to finalize a running hash computation (latency of 24 clock cycles). The buffer is managed by two pointers that grow in opposite directions. One of these pointers is always active and available to index and store the bitstream being generated, and the second is a spare pointer that may be used to index the previous bitstream until the corresponding hash computation is completed. When the active pointer reaches its maximum index ($\mathtt{B_{CF}}$ length), it switches status to a spare pointer until $\mathtt{B_{CF}}$ is ready to be read out. The bitstream currently being generated is then indexed into the buffer by the other pointer which is currently the active one. Whenever $\mathtt{B_{CF}}$ is read out (after the corresponding $\mathtt{H_{DF}}$ is generated),

its indexing pointer is reset and remains available and idle until it is activated again. The pointers switch roles in a round-robin fashion to guarantee continuous recording of all control-flow events. The buffer capacity is configured such that the bitstream indexed by the spare pointer is always read out (after the hash computation completes) before the active pointer overwrites the previous bitstream.

Load/Store Instruction Tracing and Buffering. The load/store instruction tracer tightly interfaces with the load/store stage of the processor core to extract the relevant information on all data memory access events. It extracts every load/store instruction that is executed (and successfully granted access to the requested data), the instruction address (in program memory), and the data memory address it accessed. For our RISC-V prototype, the Instruction Set Architecture (ISA) supports a 32-bit byte-addressable address space and eight load/store instruction variants. These instructions can access either a full word (four bytes), half word (two bytes), or a single byte of data memory. The number of bytes accessed is indicated by the `funct3` bits in the instruction itself, and does not need to be explicitly recorded by LiteHAX in $\mathtt{H}_{\mathtt{DF}}$. The extracted data is buffered into a dedicated FIFO buffer. It is read by the hash controller and forwarded, in turn, along with the relevant control signals to the cryptographic hash module whenever the latter is ready to absorb input. The capacity of this buffer is configured such that it would prevent dropping of any data-flow events due to latency incurred by the hash module as described below.

Cryptographic Hash Module and Controller. Our scheme requires that a single hash computation runs over a variable number of data-flow events. We employ a SHA-3 512-bit high-throughput open-source hash module [6] operating at a maximum clock frequency of 150 MHz. It comprises a padding module and permutation module which operates on a message block size of 576-bit. Input is absorbed by the core first into a padding module to assemble the 576-bit block size. Once this padding is full, the permutation module begins computation on the assembled block. In the meantime, the padding module starts assembling the next message block. The high-throughput core is configured to absorb a 64-bit input every clock cycle from the load/store instruction memory into the padding module for the duration of 9 clock cycles. After this the 576-bit buffer becomes full and notifies the permutation module to begin its computation. Once full, the padding buffer cannot absorb further input for only three clock cycles after which it resumes absorption normally.

A single data-flow event is 96-bit consisting of a 32-bit instruction address, 32-bit instruction encoding, and 32-bit data memory address. Every two consecutive data-flow events are buffered to generate three 64-bit inputs to the hash core. This effectively reduces the hash computation per data-flow event. In case the encoding buffer is full at an odd number of data-flow events, the last hash input is zero-padded. Using this hash core, a streaming input of a variable number of data-flow events can be continuously hashed until the core is signaled to conclude the current computation. Finalizing a current hash computation incurs a maximum latency of 24 clock cycles before a new hash measurement is re-initialized and the hash core can absorb input again from the load/store instruction buffer. Hence, the load/store instruction buffer is configured to guarantee that data-flow events that arrive during these cycles, either when the padding buffer cannot absorb further input or when the hash computation is being concluded, are not dropped. They are buffered until the module is ready again. A hash controller is implemented to regulate the operation of the hash module and synchronize it with other modules of LiteHAX that are responsible for the control flow tracing.

[6] http://opencores.org/project,sha3

5.2 Verifier Implementation

To prototype the verifier $\mathcal{VRF}$, we use and build on top of a Python framework, called *angr*[7], which provides capabilities to perform static binary analysis and dynamic symbolic execution mechanisms for multiple architectures. Recall that $\mathcal{VRF}$ has access to the source code of the program `Prog` being attested, and it receives the control-flow transfers executed by $\mathcal{PRV}$. Offline, $\mathcal{VRF}$ performs static and dynamic analysis to generate the CFG. Online, it uses the CFG and runs a runtime context-sensitive analysis constrained by the received control-flow execution trace. Accordingly, it performs a context-sensitive control-flow integrity check for each control-flow transfer of the received execution trace and sequentially activates validated transfers and basic blocks in the CFG.

Simultaneously, $\mathcal{VRF}$ traverses and analyzes the activated basic blocks of the CFG by a context-sensitive forward data-flow analysis. While iterating through its instructions, it generates state changes including objects accessed by each store/load instruction. It also runs a path-constrained symbolic execution to generate the executed instructions and the memory addresses they accessed. $\mathcal{VRF}$ compares the memory accesses obtained from the symbolic execution with the data-flow analysis results. Besides providing a more accurate path- and context-sensitive analysis, knowing the execution path minimizes the computational effort and eliminates the exponential explosion in analysis time by constraining analysis and symbolic execution to a single path.

6 LiteHAX: Evaluation & Security

We prototyped LiteHAX to target bare-metal embedded software executing on a single-core processor architecture. For the prover, we implemented the hardware modules of LiteHAX in Verilog, interfaced it with the open-source RISC-V Pulpino core [8], simulated and synthesized it. We evaluated the functionality and performance of LiteHAX using Open Syringe Pump [9], an open-source embedded syringe pump application and CoreMark [10]. For the verifier, we used the *angr* framework for the offline CFG generation analysis and online context-sensitive analysis and validation for Open Syringe Pump and CoreMark. We discuss next our evaluation results.

6.1 Prover

Functionality. We extended the Pulpino RTL with LiteHAX and performed cycle-accurate simulation on ModelSim while executing the aforementioned programs. We confirmed that LiteHAX extracts and reports all control-flow and memory load/store instructions and that none are dropped even when the $\mathtt{B}_{\mathtt{CF}}$ buffer is full and when the hash module is not ready to absorb subsequent instructions.

Performance, Area and Memory. Synthesis and implementation results using Xilinx Vivado indicate that LiteHAX can be clocked at a maximum frequency of 150 MHz on an Artix-7 XC7Z020 FPGA device on a Zedboard, well above that of Pulpino and on par with the SHA-3 engine we use. The LiteHAX modules ensure that zero performance overhead is incurred on the executing program being attested. LiteHAX consumes 2% of the available registers and 3% of available LUTs on an Artix-7 XC7Z020 FPGA device which amounts to ≈15% additional overhead to the base Pulpino SoC.

The memory utilization depends on the configuration of the control-flow bitstream buffer and the load/store instruction memory.

[7] http://angr.io/
[8] https://github.com/pulp-platform/pulpino
[9] https://hackaday.io/project/1838-open-syringe-pump
[10] https://www.eembc.org/coremark//index.php

We configure the load/store instruction memory to buffer up to 32 memory access operations which consumes 3Kbit of distributed RAM. While the configuration of the control-flow encoding buffer is limited by the memory resources on the device, other metrics are also affected. A smaller buffer consumes less on-chip memory and reduces the latency in detecting individual runtime attacks but also increases the overhead incurred in transmitting more attestation report packets and initiating the online verification process more often (cf. § 6.2 for online verification phase runtime). For prototyping, we configured the control-flow buffer to consume 35 36Kb BRAM blocks which is used to encode around 40,000 control-flow transfers assuming they are all, at worst case, indirect jumps and require 32-bit encoding. In practice, this is entirely code dependent but we observe that only around 44% of all control-flow transfers in the Syringe Pump are indirect jumps. The control-flow buffer is also sized to absorb incoming control-flow events when the hash module is busy finalizing the previous hash computation.

6.2 Verifier

We evaluated the verifier $\mathcal{VRF}$ performance by measuring the runtime of the online verification phase of LiteHAX for Syringe Pump when dispensing three different amounts of liquid. The reference CFG is generated once in an offline phase, while in the online phase, $\mathcal{VRF}$ re-constructs the control-flow path executed and checks that each control-flow transfer is allowed according to the CFG. It simultaneously runs symbolic execution and data-flow analysis while traversing the basic blocks of the validated control-flow path to generate the executed and legal memory access operations respectively. Table 1 shows the verification runtime measured for this online phase and how it is directly proportional to the number of control-flow transfers. We also measured the online phase runtime of CoreMark for an execution path consisting of 7,947,881 basic blocks to be 29,738 seconds. Our measurements were conducted on a system with Linux OS, 39 GB of system memory, and Intel Haswell Core Processor @ 2.2GHz.

Table 1: Verifier runtime measurements of online phase for Syringe Pump.

Liquid Quantity	Control-Flow Transfers	Runtime (s)
0.1 ml	6333	57
0.2 ml	12480	115
0.5 ml	30912	283

6.3 Security Considerations and Limitations

To enable detection of control-flow hijacking and data-oriented attacks by $\mathcal{VRF}$, LiteHAX is required to provide a *continuous*, *accurate*, *complete*, *authentic*, and *fresh* attestation of the control and data flow of a program execution at $\mathcal{PRV}$. Recall that the adversary $\mathcal{ADV}$ is incapable of modifying software binary at load-time (due to static attestation) or at runtime (due to W$\oplus$X).

Runtime Attacks: Control-Flow Hijacking and DOP Attacks. LiteHAX uses hardware modules that are tightly integrated with the processor to extract and encode all control-flow information and memory access events directly from the processor pipeline. This in addition to the sufficiently sized buffers for control-flow and $\mathtt{H_{DF}}$ for load/store instructions (cf. § 6.2) guarantee that every control-flow and load/store event is received from the processor pipeline and recorded (i.e., *completeness*). The control flow is encoded such that the execution path can be accurately re-constructed by $\mathcal{VRF}$; direct jumps and conditional branches are encoded by a single bit, while for each indirect jump, the destination address is included in $\mathtt{B_{CF}}$. Since load/store instructions are the only instructions that can access memory on RISC-based architectures, they are the only operations that influence data flow, hence the only operations corrupted to perform illegal memory accesses for non-control-data and DOP exploits. Hence, tracing the data flow by recording every load/store instruction, its address and the data memory address it accesses is sufficient to detect all known DOP attacks (i.e., *accuracy*). Since LiteHAX is hardware-based, it cannot be compromised by malicious software. The on-chip memory utilized by LiteHAX is hardware-protected and not mapped to software-accessible address space, hence protected from adversarial access by software means.

Attestation Protocol and Network Attacks. Attestation occurs continuously and is coupled with code execution at $\mathcal{PRV}$ and is reset only when $\mathcal{PRV}$ is reset. The recorded memory access events are measured into a compact cryptographically-secure hash digest $\mathtt{H_{DF}} = \mathtt{hash}(\mathtt{Load}_1/\mathtt{Store}_1 \| \ldots \| \mathtt{Load}_n/\mathtt{Store}_n)$. To evade detection of data-flow attacks, finding a second image (sequence of load/store instructions) that maps to the reference hash value is required. However, that is infeasible since the hash engine used for generating $\mathtt{H_{DF}}$ is second pre-image resistant. Every intermediate attestation report is authenticated by $\mathcal{PRV}$ along with a monotonic counter `ctr` using a cryptographically-secure digital signature $\sigma = \mathsf{sign}\{sk_{\mathcal{PRV}}; \mathtt{B_{CF}} \| \mathtt{H_{DF}} \| \mathtt{ctr}\}$ based on $\mathcal{PRV}$'s signing key $sk_{\mathcal{PRV}}$. Note that, $sk_{\mathcal{PRV}}$ is stored in hardware-protected memory that is only accessible by LiteHAX. The signature and secure key storage guarantee the *authenticity* of the report while the monotonic counter ensures its *freshness*. Note that the monotonic counter is backed with non-volatile memory and is non-resettable even when $\mathcal{PRV}$ is reset to protect against network replay attacks.

Physical Attacks. While expensive invasive/semi-invasive physical attacks are generally considered out of scope in this work, most other physical attacks can be detected by LiteHAX. In particular, physical attacks that aim at manipulating the program code at runtime as well as fault injection attacks will be detected by $\mathcal{VRF}$ if they affect the control flow or data flow or memory access instructions of the program. To evade detection of code modification at runtime by LiteHAX, the adversary is restricted to mounting an expressive attack while preserving both the control flow or data flow. We chose to also include the memory access instructions in the hash computation (while unnecessary for mitigating control-flow and data-only runtime attacks) to make it increasingly difficult in practice for an adversary to evade detection of code modification.

Practical Limitations. Effectively detecting all runtime attacks relies largely on the analysis and verification techniques deployed by $\mathcal{VRF}$. The context-sensitive analysis we deploy in this work detects attacks that break context-sensitivity and memory corruptions across data objects. However, detecting memory corruptions across fields within the same object would require sophisticated field-sensitive data-flow analysis at $\mathcal{VRF}$ which is out of scope. If detecting attacks with minimum time latency is required, the online verification at $\mathcal{VRF}$ would need to keep up with the actual execution which is not possible if they both run at the same clock speed. However, we assume that, in a practical setting, $\mathcal{VRF}$ operates with the maximum available computation resources while the actual execution would be running on an embedded device clock frequency, which would compensate for the difference. Finally, symbolic execution theoretically may fail to successfully resolve a symbolic expression required to generate data memory accesses. However, we have not encountered such case in our evaluation of Open Syringe Pump and CoreMark. However, this can be identified either at runtime or a priori by offline analysis and flagged.

7 Related Work

Static Attestation. Attestation is used to allow a third party (verifier) to check the trustworthiness of a remote device (prover).

Many approaches to remote attestation have been proposed in literature. These are classified into: (i) Software-based attestation schemes [15, 24, 27] that allow attestation of legacy low-end embedded devices and require no hardware support or cryptographic secrets. These provide weak security guarantees [32]. (ii) Hardware-based attestation schemes [21, 31] that provide stronger security guarantees based on a secure co-processor such as TPM. These are, however, complex for low-end embedded devices, and (iii) Hybrid attestation schemes [3, 4, 12, 19] that allow scalable attestation of embedded devices while requiring minimal hardware security features. Static attestation only measures program binaries at load-time and is incapable of detecting runtime attacks.

Mitigation of Runtime Attacks. Many defenses have been proposed over the past few years to mitigate runtime exploits [30]. Control-flow integrity (CFI) [1] ensures that a program follows a valid path in its control-flow graph (CFG). However, CFI cannot mitigate non-control-data and DOP attacks. Fine-grained code randomization [23] randomizes the code layout at different granularities. However, an attacker can still exploit a branch instruction to jump to the target address of choice and mount a runtime attack [17]. Code-Pointer Integrity (CPI) [22] aims at ensuring the integrity of code pointers but it also does not mitigate non-control-data attacks.

Runtime Attestation. To detect some non-control-data runtime attacks, control-flow attestation (C-FLAT [2]) was recently proposed. C-FLAT enables reporting the exact control-flow path of an executing program to a trusted verifier that can be assured of the program's execution. However, C-FLAT requires code instrumentation and incurs high runtime overhead due to frequent hash computations and context switching at the prover. It is also not scalable at the verifier that is required to search a large database of expected execution paths. LO-FAT [10] and ATRIUM [33] leverage hardware extensions to record and measure control-flow events during program execution and thus, eliminate the performance overhead of C-FLAT. ATRIUM additionally tracks executed instructions to detect memory manipulation attacks by a stronger adversary. However, all of the above runtime attestation schemes can only detect control-flow and some non-control-data attacks but cannot detect DOP attacks.

DOP Attacks and Mitigation. DOP attacks corrupt non-control-data to stitch together a sequence of operations to perform highly expressive execution on attacker-controlled input without modifying the control flow of the executing program [13, 14]. DOP attacks continue to be a challenging class of attacks to effectively mitigate. Existing defenses against DOP attacks (e.g., memory safety [11, 25], data-plane randomization [5, 6], data-flow integrity (DFI) [7]) are highly inefficient or provide limited simultaneous protection domains. Hardware-assisted data scoping enforcement and isolation architectures [26, 29] either incur high area overhead or provide limited number and granularity of protection domains. Enforcing DFI in all memory regions would mitigate DOP attacks at runtime but incurs a prohibitively high overhead [7, 13]. LiteHAX enables efficient after-the-fact detection of these attacks by attesting the control-flow and data-flow events at execution while performing the data-flow analysis at verification afterwards.

8 Conclusion

Embedded devices represent an attractive target for remote attacks largely due to their pervasiveness, connectivity, and lack of appropriate defenses. Runtime attestation schemes have been proposed to detect runtime attacks but none of which effectively deter the more expressive data-oriented programming (DOP) attacks. In this work, we presented LiteHAX, an efficient hardware-assisted runtime attestation scheme that can continuously attest both control- and data-flow events of the executing program at runtime, thus, enabling detection of all runtime attacks, including known DOP attacks. LiteHAX does not require invasive micro-architecture modifications, architecture extensions or any code instrumentation. We demonstrate a proof-of-concept implementation of LiteHAX that targets a RISC-V SoC and evaluate its performance and efficiency.

Acknowledgments. This work was co-funded by the German Research Foundation (DFG) within CRC 1119 CROSSING, the German Federal Ministry of Education and Research (BMBF) within HWSec, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] M. Abadi et al. Control-Flow Integrity: Principles, Implementations, and Applications. *ACM TISSEC*, 2009.
[2] T. Abera et al. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *ACM CCS*, 2016.
[3] M. Ambrosin et al. SANA: Secure and Scalable Aggregate Network Attestation. In *ACM CCS*, 2016.
[4] N. Asokan et al. SEDA: Scalable Embedded Device Attestation. In *ACM CCS*, 2015.
[5] S. Bhatkar et al. Data Space Randomization. In *DIMVA*, 2008.
[6] C. Cadar et al. Data Randomization. Technical report, 2008.
[7] M. Castro et al. Securing Software by Enforcing Data-flow Integrity. In *OSDI*, 2006.
[8] S. Chen et al. Non-Control-Data Attacks Are Realistic Threats. In *USENIX*, 2005.
[9] F. B. Cohen. Operating System Protection through Program Evolution. *Computer & Security*, 1993.
[10] G. Dessouky et al. LO-FAT: Low-Overhead Control Flow ATtestation in Hardware. In *DAC*, 2017.
[11] J. Devietti et al. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *ASPLOS*, 2008.
[12] K. Eldefrawy et al. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS*, 2012.
[13] H. et al. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *IEEE S&P*, 2016.
[14] C. Evans. Advancing exploitation: a scriptless 0day exploit against linux desktops, 2016.
[15] R. Gardner et al. Detecting Code Alteration by Creating a Temporary Memory Bottleneck. *IEEE TIFS*, 2009.
[16] Hewlett-Packard. Data Execution Prevention, 2006.
[17] S. Hovav et al. On the Effectiveness of Address-space Randomization. In *ACM CCS*, 2004.
[18] Intel. Control-flow Enforcement Technology Preview, 2016.
[19] P. Koeberl et al. TrustLite: A Security Architecture for Tiny Embedded Devices. In *EuroSys*, 2014.
[20] T. Kornau. Return Oriented Programming for the ARM Architecture. Master's thesis, Ruhr-University Bochum, 2009.
[21] X. Kovah et al. New Results for Timing-Based Attestation. In *IEEE S&P*, 2012.
[22] V. Kuznetsov et al. Code-pointer Integrity. In *OSDI*, 2014.
[23] P. Larsen et al. SoK: Automated Software Diversity. In *IEEE S&P*, 2014.
[24] Y. Li et al. VIPER: Verifying the Integrity of Peripherals' Firmware. In *ACM CCS*, 2011.
[25] S. Nagarakatte et al. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*, 2009.
[26] T. Nyman et al. HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement. *CoRR*, 2017.
[27] A. Seshadri et al. SWATT: Software-based Attestation for Embedded Devices. In *IEEE S&P*, 2004.
[28] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM CCS*, 2007.
[29] C. Song et al. HDFI: Hardware-Assisted Data-Flow Isolation. In *IEEE S&P*, 2016.
[30] L. Szekeres el al. SoK: Eternal War in Memory. In *IEEE S&P*, 2013.
[31] Trusted Computing Group (TCG). Website. http://www.trustedcomputinggroup.org, 2015.
[32] G. Wurster et al. A Generic Attack on Checksumming-based Software Tamper Resistance. In *IEEE S&P*, 2005.
[33] S. Zeitouni et al. ATRIUM: Runtime Attestation Resilient Under Memory Attacks. In *ICCAD*, 2017.

D

HARDSCOPE: HARDENING EMBEDDED SYSTEMS AGAINST DATA-ORIENTED ATTACKS

[105] 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 IEEE/ACM Design Automation Conference (DAC). ACM/IEEE, 2019. Core Rank A.

HardScope: Hardening Embedded Systems Against Data-Oriented Attacks

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Ghada Dessouky
Technische Universität Darmstadt, Germany

Shaza Zeitouni
Technische Universität Darmstadt, Germany

{ghada.dessouky,shaza.zeitouni}@trust.tu-darmstadt.de

Aaro Lehikoinen
Aalto University, Finland
aaro.j.lehikoinen@aalto.fi

Andrew Paverd
Aalto University, Finland
andrew.paverd@ieee.org

N. Asokan
Aalto University, Finland
asokan@acm.org

Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Memory-unsafe programming languages like C and C++ leave many (embedded) systems vulnerable to attacks like control-flow hijacking. However, defenses against control-flow attacks, such as (fine-grained) randomization or control-flow integrity are ineffective against data-oriented attacks and more expressive *Data-oriented Programming* (DOP) attacks that bypass state-of-the-art defenses.

We propose *run-time scope enforcement* (RSE), a novel approach that efficiently mitigates *all currently known DOP attacks* by enforcing compile-time memory safety constraints like variable visibility rules at run-time. We present Hardscope, a proof-of-concept implementation of hardware-assisted RSE for RISC-V, and show it has a low performance overhead of 3.2% for embedded benchmarks.

1 INTRODUCTION

Data-oriented attacks can influence program behavior without the need to modify control-flow data. Instead, they corrupt variables used by the program's decision making, or leak sensitive information from program memory. Such attacks are called non-control-data attacks [7]. Non-control-data attacks have been shown to allow attackers to forge user credentials, change security critical configuration parameters, bypass security checks, and escalate privileges. Recent work shows that it is even possible to generalize data-oriented attacks to construct full-blown malicious attacks with Turing-complete expressiveness, called *Data-Oriented Programming* (DOP) [15]. Such attacks are executed by carefully corrupting only non-control data over time to chain together sequences of operations on attacker-controlled input. DOP provides similar capabilities to attackers as return-oriented programming [26], but without breaking the victim program's control-flow integrity. This, combined with the ability for DOP to reuse virtually any data, makes preventing DOP attacks a significant and open challenge.

Existing defenses against control-flow attacks cannot prevent data-oriented attacks. Some defenses against non-control-data attacks (e.g., [5, 24]) protect individual pieces of (security-critical) data. Hu et al. [15] discuss various existing schemes that could reduce the number of DOP attacks, including memory safety, data-flow integrity, fine-grained data-plane randomization, and hardware/software fault isolation. However, they explain that existing approaches are either too coarse grained, or result in prohibitively high performance overheads. Without viable alternatives, and because effective defenses against control-flow attacks are already being deployed, DOP is likely to become the next appealing attack technique for run-time exploitation.

Goals and Contributions. We propose a new *efficient* defense against data-oriented attacks that effectively prevents *all currently known* DOP attacks. It can also be configured to prevent control-flow hijacking. The intuition behind our approach is simple: In *block structured languages* every variable has a *lexical scope*, denoting the block(s) of source code in which the variable is visible. All correct compilers enforce *variable scope* at compile-time by checking these variable visibility rules. All currently known DOP attacks, and many data-oriented attacks in general, violate variable scope rules at run-time, since there is no equivalent enforcement. Consequently, mechanisms for variable scope enforcement *at run-time* can significantly reduce the exposure to data-oriented attacks.

In this paper, we define the notion of *Run-time Scope Enforcement* (RSE) that provides *fine-grained compartmentalization* of data memory within programs. We then describe HardScope, a *hardware-assisted* RSE scheme. HardScope differs from existing defenses in the following important ways: a) it provides *complete meditation* of all variables accesses, b) it is *efficient*, incurring only a small performance overhead for embedded benchmarks, and c) it enables *context-specific* policies. This means that the same piece of code can be granted access to different memory locations depending on the context in which the code is executed. Our main contributions are:

- *Run-time Scope Enforcement*: A novel approach for fine-grained **context-specific memory isolation** within programs (Section 3) to defeat data-oriented attacks.

An extended version of the work available [22].

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317836

- *HardScope*: An open-source proof-of-concept implementation of hardware-assisted RSE on the RISC-V architecture that demonstrates **efficient memory compartmentalization** (Section 4).
- *Compiler support and APIs*: Compiler support for **protecting static and automatic variables at run-time** (Section 4.3) without requiring any developer input, and a **programmer's API** (Section 4.4) that allows developers to annotate dynamic allocations to complement the automated instrumentation.
- *Evaluation*: Analysis of RSE security guarantees (Section 5.1), and evaluation of HardScope's hardware area overhead and minimal performance impact (Section 5.2).

2 ADVERSARY MODEL & CHALLENGES

Adversary Model. We consider a powerful adversary who has full control over the data memory of the target program. This models buffer overflows and other memory corruption vulnerabilities (e.g., an externally controlled format string[1]) that could corrupt any data memory. However, the adversary cannot modify program code (W⊕X protection). Our adversary model is standard for run-time attacks and consistent with Hu et al.'s DOP attacks [15].

Challenges. Our goal is to prevent the above adversary from mounting DOP attacks. Since DOP attacks (similar to many other data-oriented attacks) require the adversary to modify and access data in unintended ways at run-time, these attacks can be prevented by a *run-time enforcement mechanism* that prevents any data access that would not be permitted during a compile-time check by a correct compiler. Designing a solution to meet this goal requires addressing the following significant challenges:

- C1 *Run-time enforcement:* enforcing variable scopes at run-time requires information which is usually only available at compile-time.
- C2 *Multi-granularity enforcement:* the enforcement mechanism must be configurable for any granularity of protection domain (subject) and protected region (object).
- C3 *Context-specific enforcement:* enforcing different permissions on each invocation of the same subject (e.g., each function), to minimize the attack surface following the principle of *least privilege*.
- C4 *Complete mediation:* protection domains cannot be allowed to increase their permissions accidentally or maliciously, and all memory accesses must be checked with only minimal performance impact and memory overhead.

3 DESIGN OVERVIEW

The high-level idea of HardScope is to extend the compiler to emit compile-time information about the visibility of variables, and to extend the underlying hardware to use this compiler-supplied information to dynamically create and update a set of memory access rules against which all memory accesses are checked.

Run-time enforcement. Machine code produced from languages such as C and C++ does not include information available to the compiler about variables and code blocks (C1). RSE needs this information to assign in-memory variables to specific *execution contexts*. To bridge this gap between compile-time lexical scope and run-time execution context, we modified the compiler to instrument the program code with special instructions that record which variables may be used by each code block. HardScope introduces an instruction set extension for this purpose (Section 4).

The compile-time components and behavior of HardScope are illustrated in Figure 1. An unmodified source code program (❶) is fed to the compiler (❷), which checks (as usual) that all variable accesses are correctly scoped. Our new *RSE Plug-in* (❸) in the compiler adds HardScope instructions (❹) at particular locations in the binary (e.g., at the start of functions). This results in a fully-functional program binary, instrumented with HardScope instructions that the HardScope hardware uses to create a set of rules against which all memory accesses can be checked at run-time.

Multi-granularity enforcement. We chose function-level compartmentalization as the granularity of isolation, since this is sufficient to mitigate all currently known DOP attacks (Section 5.1). However, RSE can also be implemented at other granularities (Section 4), without changes to the new HardScope hardware (C2).

Context-specific enforcement. Consider the program (❶) in Figure 1: function `C` receives two pointers and copies data from the first pointer to the second. It can be called from either function `A` or function `B` (call graph shown in Figure 2). In benign execution, variables x and y are only used in a *privileged* execution path, where access control checks prevent misuse (e.g., x could be a secret key). Function `B` contains an exploitable vulnerability allowing the attacker to control the pointers passed to function `C`. Since function `C` can be used to copy arbitrary data between two attacker-controlled pointers, this constitutes a DOP gadget. The attacker could use this to bypass the access control checks on variables `x` and `y` by accessing them through the unprivileged execution path.

HardScope prevents this by providing context-specific enforcement, in which different memory access rules can be associated with *each active instance* of a function (C3). To achieve this, the HardScope hardware creates memory access rules dynamically for each individual function invocation, and stores these in a data structure called the *Storage Region Stack* (SRS). The SRS is kept in hardware-isolated *protected memory*; only HardScope instructions can add or remove SRS entries. Each SRS entry defines an area of memory (e.g., the location of a variable) that may be accessed. The SRS is organized into *frames*; each frame contains all the entries for a particular execution context. The topmost SRS frame corresponds to the active execution context. On each memory access, e.g., load or store, HardScope validates that the memory address matches an entry in the topmost SRS frame.

Specifically, HardScope prevents the attack in Figure 2 as follows: The SRS for function `A` (❺) includes variables x and y, and the SRS for function `B` (❻) includes variables i and j (Figure 2). To allow function `C` to access certain variables, the calling function must use a special instruction (Figure 1 ❼) to *delegate* access to a variable to function `C`: e.g., function `A` must delegate access to x and y. For valid delegation, the calling function must already have access to the delegated variables. Even though the attacker can still manipulate the pointers in function `B`, this function does not have access to x and y (no corresponding SRS entries) and hence it cannot delegate access to these variables to function `C`.

[1] CWE-134: Use of Externally-Controlled Format String
https://cwe.mitre.org/data/definitions/134.html

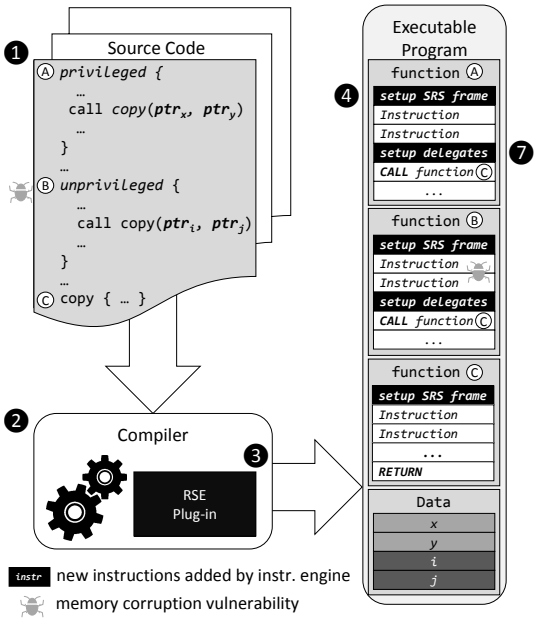


Figure 1: Compile-phase design of HardScope. Run-time memory accesses via pointers ptr_x, ptr_y are limited to variables x and y, while ptr_i, ptr_j are limited to i and j.

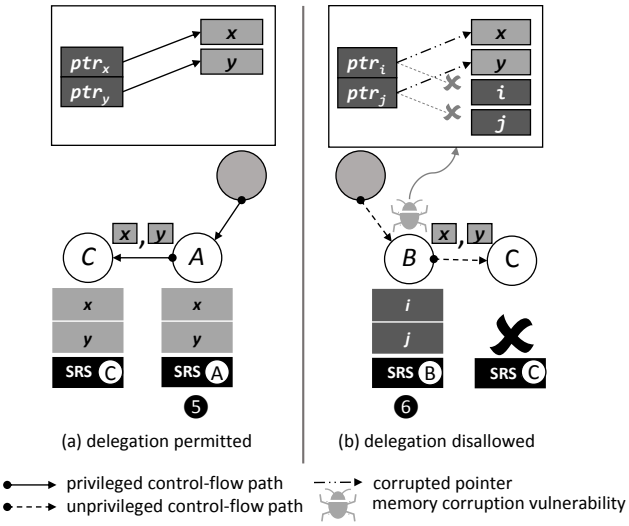


Figure 2: Run-time design of HardScope showing the call graph of program in Figure 1. In (a), access to variables x and y is successfully delegated from A to C. In (b), function B should not have access to x and y, but a memory corruption vulnerability in B is used to corrupt ptr_i and ptr_j to point to x and y instead of i and j. HardScope prevents B from accessing or delegating x and y.

4 IMPLEMENTATION

We developed a proof-of-concept hardware implementation of HardScope and integrated it into the open-source RISC-V Pulpino core.[2] HardScope extends the RISC-V instruction set with seven new SRS management instructions, as shown in Table 1. We augmented the GCC compiler to incorporate a proof-of-concept RSE plug-in and a modified RISC-V backend to automatically instrument C programs with the relevant HardScope instructions. These protect static and automatic variables at run-time without requiring any changes to program code. We also developed a HardScope **Programmer's API** (Section 4.4) that allows developers to annotate dynamic allocations to complement the automated instrumentation. HardScope itself is architecture-agnostic; our choice of RISC-V and Pulpino is due to the open-source nature of the ISA and the RTL implementation, thus enabling us to prototype our solution.

4.1 Instructions

The `sbent` and `sbxit` instructions are used to mark the beginning and end of each execution context. HardScope uses these instructions to track when HardScope is first enabled and when the execution context changes, and thus when new enforcement rules should be loaded in the SRS. `sbent` pushes a new frame on top of the SRS, whilst `sbxit` pops the topmost SRS frame. Program execution starts with an empty SRS and HardScope enforcement is initially disabled. HardScope is enabled by the first `sbent`, and remains enabled until a matching `sbxit` empties the stack.

The `sradd` and `srdda` instructions create an SRS entry in the current (topmost) SRS frame. HardScope uses these instructions to determine the bounds of memory areas that the current execution context is allowed to access. The two operands set the *base* and *limit* address of the storage region respectively. An optional offset is added to to either the limit (`sradd`) or base (`srdda`) register operand.

The `srdel` instruction removes the specified number of SRS entries from the current SRS frame (last in first out). It allows the program to drop unneeded memory access privileges without changing execution context.

The `srdlg` and `srdsub` instructions delegate an SRS entry from the currently executing function either to an invoked callee function or to the caller when the current function returns. HardScope uses these instructions to derive SRS entries for data flows which are not known at compile-time, such as context-specific accesses (Section 3). The operands specify an address to determine which memory address to delegate. The resulting memory address is compared with the current SRS entries and if a match is found, the most recent matching entry is copied to the *next execution context entered*. If the delegation is followed by a `sbent`, the delegated entry is added to the newly created SRS frame. If the delegation is followed by a `sbxit`, the delegated entry is added to the caller's SRS frame.

The `srdsub` instruction is used to delegate a new SRS entry that is a subset of an existing SRS entry. It takes the same operands as `sradd`. If the new subdivided memory region is a subset of an existing SRS entry in the current SRS frame, a new SRS entry is created for a *sub-region* using the new base and limit.

If no matching entry is found in the SRS when `srdlg` or `srdsub` execute, no entry is delegated. This prevents the use of `srdsub` to elevate the access rights of the next execution context beyond the rights of the current, but allows the delegation instructions to be applied to pointers which are not dereferenced directly in the current context. These include null-pointers and intentionally created out-of-scope pointers (e.g., via the use of pointer arithmetic)

[2] http://www.pulp-platform.org/

Table 1: HardScope Instructions. *Operands* lists valid combinations of operands: `r`*n* is a register, `imm` is an immediate value, and `imm(r`*n*`)` is a register to which an immediate offset is added. *Cycles* indicates the number of cycles consumed at execute stage.

Mnemonic	Name	Operands	Cycles
`sbent`	**s**cope **b**lock **ent**er	n/a	1 (+ *N*)
`sbxit`	**s**cope **b**lock e**xit**	n/a	1 (+ *N*)
`sradd`	**s**torage **r**egion **add**	`r1, imm(r2)`	1
`srdda`	**s**torage **r**egion **dda** (reverse add)	`imm(r1), r2`	1
`srdel`	**s**torage **r**egion **del**ete	`imm(r1)` `imm`	1 (+ 1)
`srdlg`	**s**torage **r**egion **dele**gate	`imm(r1)` `imm`	1 (+ 1)
`srdsub`	**s**torage **r**egion **d**elegate **sub**-region	`r1, imm(r2)`	1 (+ 1)

that are passed to callees for which they are in scope (e.g., accessor functions that receive opaque pointers as arguments).

4.2 Hardware Implementation

We modified the instruction decoding stage of the processor pipeline to interpret the new instructions (Section 4.1). After decoding, the appropriate control signals are sent to the HardScope unit, which realizes the execute stage of the new instructions. Figure 3 shows the main components of the HardScope unit: the *SRS controller* (❶), dedicated memory to hold the SRS (❷), and three register banks (❸, ❹, ❺). The *active bank* (❸) holds the entries in the SRS frame for the current execution context enabling each memory access to be compared against *all active entries* efficiently. The *spare bank* (❹) holds entries delegated via `srdlg` and `srdsub` before a HardScope context switch occurs. It allows delegated entries for the *next execution context* to be accumulated ahead of time. When a HardScope context switch occurs, the spare bank becomes the active bank (and vice versa), thus activating the delegated entries. The third bank (❺) is used as a cache to hold a copy of the topmost frame of the SRS. This reduces the latency when the topmost SRS frame is transferred between the stack memory and the spare bank.

When executing `sbent`, the controller activates the spare bank and transfers the contents of the currently active bank to the cache (❻) in a single cycle. The bank that held the previously active frame becomes the spare, and can be used for subsequent delegations. The entries in the cache must be stored for future use, and are transferred to the SRS in protected memory (❼) over at most *N* subsequent cycles, where *N* is the maximum number of entries in the cache. During this time, the CPU continues to execute subsequent instructions normally until a new HardScope context switch occurs. Only if a HardScope context switch occurs before the cache has been emptied does the processor stall until the transfer is complete.

When executing `sbxit`, the controller copies the SRS frame from the cache into the spare bank (❽) while retaining delegated entries (i.e., activating the entries that are already in the spare bank). The SRS frame in the previously active bank is no longer needed and is discarded. This executes in a single cycle. The cache, which now holds an out-of-date copy of the active frame, is updated with the topmost SRS frame from the protected memory (❾), which takes at most *N* cycles, where *N* is the number of entries in the topmost SRS frame in memory. This does not stall the processor unless another `sbxit` is encountered before the cache is fully populated, in which case the CPU stalls until the next frame is available. However, if

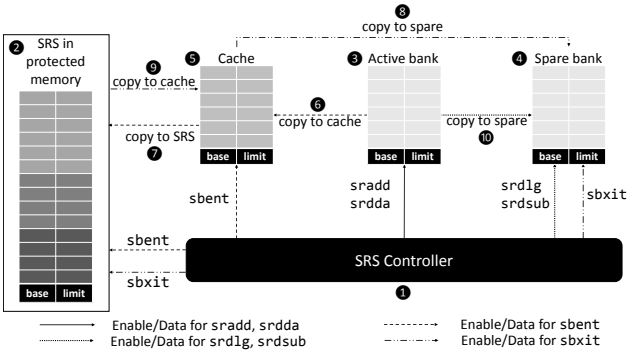


Figure 3: HardScope hardware architecture.

an `sbent` is encountered before the cache is fully populated, the partial cache is discarded and replaced with the contents of the active bank, without stalling.

The `sradd` and `srdda` instructions always operate on the active bank. When executing `srdsub`, the controller checks the active bank for an entry containing the given memory region and, if found, adds the new sub-entry to the spare bank. Similarly, in `srdlg`, the controller checks for the matching entry in the active bank and, if found, copies the entry to the spare bank (❿). The `srdlg` and `srdsub` instructions require an additional cycle only if followed immediately by a `sbent` or `sbxit`.

Integrating HardScope into the processor pipeline also required modifying the memory access stage to intercept all memory access requests to the load/store unit. At each `load` or `store` instruction, the requested memory address and the number of requested bytes (one byte, half-word (two bytes), or word (four bytes)) are intercepted and forwarded to the SRS controller, which compares it against all entries in the active bank. The registers in each bank are wired to comparators such that all entries in the bank are checked *in parallel*. If a match is found, i.e. the requested address range is a subset of any of the active entries, then the memory access is granted by the processor's load/store unit, otherwise a hardware fault exception is raised. We design and integrate HardScope to the processor pipeline such that no additional clock cycle latency is incurred to the baseline `load` and `store` instructions.

4.3 Software Instrumentation

Our RSE GCC plug-in and the modified RISC-V backend currently supports automatic instrumentation of C programs at *function granularity* to protect the 1) call stack frame including local variables, return address and other return state information, 2) arguments passed on the stack, 3) heap objects, and 4) global and static variables. The beginning of each distinct execution context is marked by inserting a single `sbent` instruction at the function call site just before the jump instruction. The end of an execution context is marked by inserting an `sbxit` instruction just before the return in the callee function. In Section 5 we show that function-level isolation is sufficient to mitigate all currently known DOP attacks. However, RSE can also be implemented at other granularities, without changes to the HardScope instructions, by inserting `sbent` and `sbxit` instructions around the instructions that comprise a distinct execution context.

4.4 HardScope Programmer's API

Deeply Nested Pointers. The HardScope Programmer's API enables the handling of code that uses deeply nested pointers e.g., traversing linked lists. This type of code is a challenge for automated instrumentation because e.g., passing the head of a linked list to a function that iterates through the list would require delegation of an SRS entry for each element of the list. Since the number of SRS entries (per frame) is constrained by the HardScope hardware (see Section 4.2), this leads to suboptimal use of HardScope hardware resources and an increased cost in HardScope context switches due to more frequent stalls at run-time. Instead, we propose a programming pattern using the HardScope Programmer's API where one `sradd` instruction is added before the dereference of member pointers to linked member elements, and one `srdel` is added after the dereference. This enables effective yet secure traversal of linked lists and other data structures containing nested pointers.

Heap object allocation. We implemented a wrapper on top of the C standard library `malloc()` function that creates SRS entries for heap allocations, and delegates these to the caller. Other library functions can be similarly wrapped to allow HardScope-instrumented code to be linked against uninstrumented libraries.

5 EVALUATION

HardScope meets the stated challenges (Section 2) as follows:

C1 **Run-time enforcement.** The RSE GCC Plug-in infers and emits the necessary HardScope instructions to manage the SRS for stack and global data, as well as dynamic allocations that follow a well-defined pattern. The HardScope Programmer's API allows handling code that is not automatically instrumentable, e.g., uses deeply nested pointers.

C2 **Multi-granularity enforcement.** HardScope can enforce policies with either coarser or finer granularity of execution contexts with the appropriate instrumentation (C2). For instance, HardScope can isolate the function prologue and epilogue from the function body, and protect return addresses on the stack from memory errors in the function body to prevent control-flow hijacking.

C3 **Context-specific enforcement.** In HardScope, the active SRS entries can differ between different invocations of the same subject, depending on which entries have been delegated to this subject (e.g., variables passed to a function by its caller or callee).

C4 **Complete mediation.** HardScope hardware checks every memory access against the active set of SRS entries; accesses without matching entries will fail. Therefore only compiler-admissible memory accesses are allowed.

Instructions that create rules at run-time could potentially be used as *confused deputies*. In a *confused deputy attack*, the attacker attempts to subvert the RSE property by misusing existing HardScope instructions at run-time to create unintended rules. Our design ensures that no such instructions are available to the attacker. Rules for static allocations (stack and global variables) are encoded directly into the instructions. Since these cannot be modified at run-time, they cannot be used as confused deputies.

Instructions that create rules that are determined at run-time are found within memory allocators, e.g., `malloc()`, or code that deals with deeply nested pointers, e.g., iterators annotated using the HardScope Programmer's API. It is reasonable to assume that memory allocators are trusted (or at least that allocations are not influencable by the attacker). We recommend that manually annotated code is vetted for allocators that create rules at run-time. Furthermore, an attacker can only initiate a confused deputy attack if he already controls some part of the code, which is very difficult since every memory access in the instrumented program is checked by the HardScope hardware.

5.1 Security Evaluation

We replicated the DOP attack by Hu et al. [15] and ported the code to Pulpino to evaluate the effectiveness of HardScope. Although it was not possible to port the complete victim ProFTPD server to our FPGA testbed, we focussed on the vulnerable `sreplace()` function [15]. All enforcement rules in our experiments are derived *without any developer annotations* – the GCC intermediate representation contains all information necessary for compile-time instrumentation, including: stack-frame sizes, global variable accesses, function calls, parameters, and return values. Function-granularity isolation is sufficient to prevent the attack.

We verified experimentally *four* ways in which RSE prevents this DOP attack: 1) it prevents the initial memory violation in `sreplace()` as it enforces the indended bounds of input and output buffers when operated on by an unsafe string copy operation (`strncpy()` with incorrect buffer length), 2) it prevents the attack from keeping internal state in unused areas of the program's data section, 3) it denies access to global variables which are accessed by the attack out of their normal context, 4) it denies access to static variables which should only be accessible by code wihin the same compilation unit. We discuss each of these in detail in the extended version of this article [22]. Any one of these would be sufficient to stop the attack, and thus the existence of four distinct mitigations demonstrates the effectiveness of RSE's layered defense strategy.

5.2 Performance and Area Evaluation

Performance overhead. We ran CoreMark[3], a standard performance benchmark for embedded systems, with varying iteration counts on a HardScope-augmented Pulpino synthesized on a Xilinx Zynq-7020 ZedBoard. We observed an average overall performance overhead of 3.2% compared to the execution of unmodified CoreMark on the unmodified Pulpino SoC. All instrumentation in CoreMark was automatically generated by our extended GCC compiler resulting in the binary size increasing by 11%. The number of entries required per SRS frame varied throughout execution between 1 and 23. The overall maximum SRS size was 71 entries in 11 frames, resulting in a memory overhead of 573 bytes (64 bits per entry + 4 bits per frame to record the number of entries).

Area and memory utilization. The area utilization depends primarily on the size of active, spare and cache banks (i.e., the number of entries per frame). All three banks are mapped to logic to guarantee single-cycle access parallel checking of all frame entries. The area utilization increases linearly as the number of entries configured per frame increases (for a fixed number of frames), since more entries must be checked in parallel. For a protected memory size of 8 entries × 16 frames, HardScope utilizes 4, 572 LUTs, 1, 760 registers, and one 36 kB block RAM (RAMB36). For a 32 entries ×

[3] http://www.eembc.org/coremark/faq.php

16 frames configuration (required for the CoreMark performance evaluation above), HardScope utilizes 30, 520 LUTs, 6, 362 registers, and one 36 kB block RAM (RAMB36).

6 RELATED WORK

Various software-only and hardware-assisted memory safety technologies have been proposed and/or deployed (e.g., [2–6, 8, 11, 13, 17–19, 24, 25]). We discuss approaches that aim to mitigate data-oriented attacks in detail in the extended version of this article [22].

Software-only defenses (e.g. DFI [6] and SoftBound [20]) can offer strong security guarantees, but their usefulness is limited by high performance overhead, and by requiring changes to the system software architecture. Consequently, the granularity of enforcement of deployed defenses are often relaxed in favor of improved performance. Memory-safe dialects of C (e.g., CCured [21], Cyclone [16], and Checked C [12]) retrofit C with compile- and/or run-time checks that prevent memory errors from occuring. However, such dialects only benefit programs which are modified to make use of enhanced language features, also incur considerable run-time overhead [16, 21], or preclude certain C features [12].

Hardware-assisted defenses (e.g., BIMA [19], HDFI [27], and CHERI [28]) promise to drastically improve the performance overhead compared to software-based defenses. However, recent advances in attacks against bounds-checking approaches [14] suggest that low-fat pointer schemes which enforce allocation bounds rather than object bounds, such as BIMA [19] are exploitable. On the other hand approaches that track object bounds in separate storage, e.g., Intel MPX [23], HardBound [11], are not faster nor more memory effient than sofware-based approaches. Hardware-assisted tagged memory allow efficient enforcement of memoru access policies, but unlike HardScope only support a small number of simultaneour protection domains (e.g. two domains in HDFI [27]). CHERI [28] is a hardware-assisted capability model that can support various protection models, but requires program re-engineering.

Run-time attestation schemes [1, 9, 10, 29] can only detect, but not prevent, control-flow and non-control-data attacks.

Although HardScope shares many of the same goals as the above schemes, it differs in several fundamental aspects. Compared to software-based schemes (e.g., DFI [6] and SoftBound [20]), HardScope has significantly lower overhead, does not require whole-program static analysis, and can enforce context-specific policies for individual function invocations. HardScope RSE policies can be instantiated for a large class of programs without additional input from developers (cf., YARRA [24]), or software re-engineering (cf., CHERI). HardScope reduces the metadata needed at execution time to the rules for active execution contexts. Active rules are cached in on-chip memory, to enable access checks with no overhead.

7 CONCLUSION

By implementing and evaluating HardScope, we demonstrated that RSE is an effective approach to protect against data-oriented attacks. HardScope can also enforce memory isolation at coarser or finer granularity, to enable different memory protection strategies.

We provide 1) our enhanced GCC compiler; 2) instrumented binaries of our test programs; and 3) a RISC-V simulator with support for HardScope instructions at https://goo.gl/TAjLxy.

ACKNOWLEDGMENTS

This work was supported by the German Science Foundation CRC 1119 CROSSING projects S2 and P3, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant nr. 643964 (SUPERCLOUD), Business Finland under grant nr. 3881/31/2016 (CloSer), Academy of Finland under grant nr. 309994 (SELIoT), and the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

REFERENCES

[1] Tigist Abera et al. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proc. ACM CCS '16*. 743–754.
[2] Periklis Akritidis et al. 2008. Preventing Memory Error Exploits with WIT. In *Proc. IEEE S&P '08*. 263–277.
[3] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Proc. DIMWA '08*. 1–22.
[4] Cristian Cadar et al. 2008. *Data Randomization.* Technical Report MSR-TR-2008-120. Microsoft Research.
[5] Miguel Castro et al. 2009. Fast Byte-granularity Software Fault Isolation. In *Proc. ACM SOSP '09*. 45–58.
[6] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proc. USENIX OSDI '06*. 147–160.
[7] Shuo Chen et al. 2005. Non-control-data Attacks Are Realistic Threats. In *Proc. USENIX Security '05*. 12–12.
[8] Long Cheng, Ke Tian, and Danfeng (Daphne) Yao. 2017. Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks. In *Proc. ACM ACSAC '17*. 315–326.
[9] Ghada Dessouky et al. 2017. LO-FAT: Low-Overhead Control Flow ATtestation in Hardware. In *Proc. ACM/EDAC/IEEE DAC '17*. 24:1–24:6.
[10] Ghada Dessouky et al. 2018. LiteHAX: Lightweight Hardware-assisted Attestation of Program Execution. In *ICCAD '18*.
[11] Joe Devietti et al. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proc. ACM ASPLOS '08*. 103–114.
[12] Archibald Samuel Elliott et al. 2018. Checked C: Making C Safe by Extension. In *Proc. IEEE SecDev '18*. 53–60.
[13] Úlfar Erlingsson et al. 2006. XFI: Software Guards for System Address Spaces. In *Proc. USENIX OSDI '06*. 75–88.
[14] Ronald Gil, Hamed Okhravi, and Howard E. Shrobe. 2018. There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection. In *Proc. IEEE SecDev '18*. 102–109.
[15] Hong Hu et al. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proc. IEEE S&P '16*. 969–986.
[16] Trevor Jim et al. 2002. Cyclone: A Safe Dialect of C. In *Proc. USENIX ATC '02*. 275–288.
[17] Dmitrii Kuvaiskii et al. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proc. ACM EuroSys '17*. 205–221.
[18] Volodymyr Kuznetsov et al. 2014. Code-pointer Integrity. In *Proc. USENIX OSDI '14*. 147–163.
[19] Albert Kwon et al. 2013. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Proc. ACM CCS '13*. 721–732.
[20] Santosh Nagarakatte et al. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proc. ACM PLDI '09*. 245–258.
[21] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proc. ACM POPL '02*. 128–139.
[22] Thomas Nyman et al. 2017. HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement. https://arxiv.org/abs/1705.10295
[23] Oleksii Oleksenko et al. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. https://arxiv.org/abs/1702.00719.
[24] C. Schlesinger et al. 2011. Modular Protections against Non-control Data Attacks. In *Proc. IEEE CSF '11*. 131–145.
[25] Konstantin Serebryany et al. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC '12*. 309–318.
[26] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proc. ACM CCS '07*. 552–561.
[27] C. Song et al. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *Proc. IEEE S&P '16*. 1–17.
[28] Jonathan Woodruff et al. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proc. IEEE ISCA '14*. 457–468.
[29] Shaza Zeitouni et al. 2017. ATRIUM: Runtime Attestation Resilient Under Memory Attacks.. In *ICCAD '17*.

E

CHASE: A CONFIGURABLE HARDWARE-ASSISTED SECURITY EXTENSION FOR REAL-TIME SYSTEMS

[40] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, and Ahmad-Reza Sadeghi. CHASE: A Configurable Hardware-Assisted Security Extension for Real- Time Systems. In IEEE/ACM International Conference on Computer-Aided Design. IEEE, 2019. Core Rank A.

CHASE: A Configurable Hardware-Assisted Security Extension for Real-Time Systems

Ghada Dessouky*⋆, Shaza Zeitouni*⋆, Ahmad Ibrahim*, Lucas Davi† and Ahmad-Reza Sadeghi*
*Technische Universität Darmstadt, Germany, {ghada.dessouky, shaza.zeitouni, ahmad.ibrahim, ahmad.sadeghi}@trust.tu-darmstadt.de
†Universität Duisburg-Essen, Germany, lucas.davi@uni-due.de

***Abstract*—Real-time autonomous systems are becoming pervasive in many application domains such as vehicular ad-hoc networks, smart factories and delivery drones. The correct functioning of these real-time systems is timing-critical with hard deadlines. However, although they interact with other systems and exchange inputs/outputs with the physical world, they usually lack security mechanisms, which makes them susceptible to a wide range of attacks with critical consequences. Typically, this is because security mechanisms usually violate the real-time requirements of these systems and cannot be adjusted at runtime to provide the adequate security without compromising performance.**

In this paper, we propose a consolidated runtime-configurable hardware-assisted security extension called CHASE that supports different levels of security at runtime. Depending on the desired security level and the system real-time, availability or functionality requirements, CHASE can be configured accordingly at runtime, thus enabling the calibration of the security vs. performance trade-off. We analyze CHASE's effectiveness in providing different security guarantees against various adversarial capabilities, and show how this is achieved with reasonable logic overhead and minimal performance overhead.

***Index Terms*—Remote attestation, real-time system security, runtime attacks, hardware-assisted security, runtime attestation**

I. Introduction

Real-time systems are ubiquitous in many application domains such as programmable logic controllers (PLCs), electronic control units (ECUs) and emerging domains that deploy networks of collaborative autonomous systems, e.g., vehicular ad-hoc networks, smart factories, search and rescue, and delivery robots and drones. Typically, such systems are required to perform their tasks in real time, while some may have hard and critical time deadlines with little tolerance for down time. They may also be deployed in safety-critical or non-deterministic infrastructures where their fail-safe operation is paramount. To perform their tasks, they might also be interconnected with the physical world and other devices, making them equally vulnerable as other systems to security exploits.

Security for Timing-Critical Applications. Nevertheless, these systems usually lack security protection mechanisms, leaving them exposed to a wide spectrum of attacks, such as the infamous Stuxnet[1] and more recently Triton[2]. Particularly, an attacker may violate memory integrity by exploiting a standard memory corruption vulnerability, e.g., externally-controlled format string[3] that causes buffer overflows leading to data memory corruption. By corrupting targeted control-flow information stored in the stack or the heap and overwriting code-pointers (return addresses or function pointers), an attacker can redirect the control flow of execution to cause a malicious and unauthorized effect. Such *runtime attacks* can be used to inject malicious code (code-injection attacks) or re-use already existing benign code chunks maliciously (code-reuse attacks), such as control-flow hijacking and data-oriented attacks [1], [2].

Protection mechanisms, such as control-flow integrity (CFI) [3] and control-flow attestation [4]) to mitigate or detect such attacks have been shown to incur non-negligible performance overheads. While this can be tolerated to some extent for applications without real-time constraints, it would violate the functionality requirements of real-time high-availability systems. Even defenses such as asynchronous CFI specifically designed for PLCs [5] still incur performance overhead of up to 8.3%. According to the NIST 800-82 guide on the security of industrial control systems [6], timing, safety and availability requirements must be prioritized when designing security mechanisms for these systems.

Attack Space Coverage and Reconfigurability. Moreover, existing defenses against runtime attacks each assume a particular adversary model and thus mitigate specific classes of attacks. Currently, no consolidated defense exists that can mitigate multiple classes of different attack vectors, or can be at least configured to thwart different adversarial capabilities depending on the desired security requirements and deployment environment. This is especially true for hardwired hardware-assisted security extensions [7]–[12] which cannot be upgraded or patched after fabrication. This makes system architects reluctant to deploy them, despite their advantages over software-based defenses.

Our Goal. In this work, we aim to tackle the challenges outlined with respect to attack space coverage and applicability of defenses for timing-critical applications. These challenges hinder the practical deployment of hardware-assisted security mechanisms for embedded systems in general, and for real-time safety-critical systems in particular. In doing so, we address the persistent trade-off between functionality requirements (e.g., real-time operation, safety, availability or other deployment constraints) vs. security requirements. We aim to provide a flexible means for the system designer to tune this trade-off by only incurring the corresponding performance overhead for the degree of the security guarantees required and configured at runtime.

To achieve this, we categorize the different classes of attacks that may target embedded systems. We evaluate which of these attacks can be detected on-device and which of them require more sophisticated policy-checking at a trusted third party. With this in mind, we provide a consolidated and configurable defense mechanism that can be adjusted at runtime to provide different security services (and thus different security guarantees against different classes of attacks) at the cost of different performance overheads. We enable this by leveraging a custom hardware-based extension, called CHASE, designed to operate in parallel to the actual processor. It captures and tracks the execution at runtime in a cycle-accurate and tightly coupled manner. CHASE is runtime-configurable and supports different security services to mitigate different adversarial capabilities, where

⋆Authors contributed equally to this work

[1]https://www.mcafee.com/enterprise/en-us/security-awareness/ransomware/what-is-stuxnet.html

[2]https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/triton-malware-spearheads-latest-generation-of-attacks-on-industrial-systems/

[3]CWE-134: Use of Externally-Controlled Format String https://cwe.mitre.org/data/definitions/134.html

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

its hardware can be configured at runtime to either *verify* or *enforce* control flow (including call-return matching). Verifying control flow checks on-device that a control-flow transfer is allowed *after* it is executed, without incurring performance overhead on the execution. Conversely, enforcing control flow actively checks that each control-flow transfer is allowed *prior* to its execution, which evidently comes with some performance overhead. For more sophisticated attacks that cannot be detected on-device, CHASE can be configured to capture fine-grained features of the relevant execution, and send them to a trusted third party (equipped with computational resources) for verification using more complex policies.

For timing-critical systems, CHASE can be configured to verify on-device that the control flow of execution is valid without interfering with the application run time or halting the execution in case of illegal control flow. Violation of a control flow policy is detected with minimal latency, after which CHASE reacts gracefully without compromising the safety or real-time requirements of the application. Execution is redirected to a pre-defined and isolated safe state which is application-dependent. In the meantime, a neighboring device (in case of a network of collaborative devices) and the trusted third party are notified of the impending exploit to react accordingly.

Contributions. In this work, we tackle the challenges we outline above by proposing:

- A modular hardware-assisted extension CHASE that consolidates different defenses and security services mitigating a larger attack space than existing defenses.
- Configurable security that can be adjusted at runtime depending on the security vs. functionality requirements of a given embedded application and the deployment scenario.
- A security mechanism that can verify control flow on-device with minimal latency making it suitable for timing-critical systems (with caveats).
- Proof-of-concept implementation for a RISC-V processor and an evaluation of performance overhead for the detection latency.

II. CHASE: System and Adversary Model

The intuition behind our work is two-fold. The first is that security mechanisms to date are *not configurable by design* to provide different levels of security at runtime. Configurable security is required when an embedded system is deployed in scenarios with different threat levels and different functionality requirements (fail-safety, real-time, periodicity, etc.). This is particularly a limitation of existing hardware-assisted extensions which cannot be modified or upgraded after production, rendering them hardwired to mitigate a fixed class of security threats for the entire lifetime of the encompassing system. Despite their many advantages as opposed to software-based schemes, this hinders their deployment in practice.

The second is that existing security extensions, even asynchronous Control-Flow Integrity (CFI) designed for PLCs [5], affect the application run time, often non-deterministically. This makes them unsuited for timing-sensitive systems that have no tolerance for down time or variable reaction times. This further emphasizes the need for a tuning knob that can be used to calibrate the trade-off between the security guarantees vs. application run time and performance overhead for different use cases and deployment settings.

To tackle the above challenges and provide configurable security and modular lines of defense, we propose CHASE. CHASE is the first consolidated and configurable custom hardware-assisted security extension that integrates tightly with the processor core of a remote in-field mid-end embedded device, called $\mathcal{DEV}$. CHASE captures and tracks the execution of $\mathcal{DEV}$ at runtime in a cycle-accurate manner using custom hardware that operates in parallel to the processor execution. It supports different security configuration modes where valid control flow can be either actively-enforced or attested after-the-fact, assuming control-flow policies are provisioned on $\mathcal{DEV}$. Verification can be performed either *on-device* or *remotely*. On-device verification checks control-flow transfers against control-flow policies that are provisioned on $\mathcal{DEV}$. Although on-device verification is limited to detecting explicit control-flow hijacking, it is low-latency and can mitigate an exploit within a few clock cycles as shown in § V-B.

Remote attestation, on the other hand, is used to detect more sophisticated attacks, such as non-control-data attacks, which do not directly compromise the control flow. It requires that the recorded and measured execution is reported to a trusted third party (called the moderator $\mathcal{MOD}$). $\mathcal{MOD}$ is assumed to be a significantly more computation-resourceful server than $\mathcal{DEV}$ and, thus can verify the reported execution against a more complex set of policies and heuristics, and detect data-oriented attacks. At runtime, the user can select to activate or deactivate any of these security services depending on the trade-off between the functionality requirements (e.g., how much performance overhead can be tolerated) and the presumed threat level of the deployment settings. In what follows, we describe and classify the adversarial capabilities and classes of attacks we consider in this work.

A. Adversary Model and Assumptions

Adversary Model. We consider an adversary $\mathcal{ADV}$ with varying capabilities depending on the deployment settings, and with full control over both the *program memory* and *data memory* of the target program executing on $\mathcal{DEV}$.

The different types of attacks $\mathcal{ADV}$ can launch against embedded systems can be broadly classified into:

A1 Static code manipulation (malware injection) attacks
A2 Runtime code-injection attacks
A3 Runtime control-flow attacks
A4 Runtime non-control-data attacks
A5 Runtime code manipulation attacks

$\mathcal{ADV}$ can launch static code manipulation attacks and inject malware such that modified code is loaded at start-up and executed (**A1**). $\mathcal{ADV}$ can also launch runtime attacks (**A2** - **A5**) by exploiting memory corruption vulnerabilities that cause buffer overflows leading to corruption of data memory. By corrupting control-flow information stored in the stack or heap and overwriting code-pointers (return addresses and function pointers), $\mathcal{ADV}$ can maliciously redirect the control flow of execution at runtime. In *code-injection* attacks (**A2**), the attacker places a malicious executable payload in program memory and redirects control flow to execute it. Alternatively, state-of-the-art runtime attacks exploit *code-reuse* techniques, e.g., *Return-Oriented Programming* (ROP) [2]. These attacks exploit memory corruption vulnerabilities and stitch together benign *gadgets* of code, which already reside in the program memory, in a particular sequence to build the attack payload and hijack the control flow of the program maliciously. These attacks hijack the control flow of the program by executing invalid control-flow transfers, that do not exist in the control-flow graph (CFG) of the program (**A3**).

More sophisticated code-reuse attacks known as non-control-data attacks (**A4**) do not explicitly compromise the control flow of a program, but cause malicious execution by corrupting critical data variables such as an authentication variable or loop variable. This

results in executing a privileged (unintended) but permissible control-flow path that exists in the CFG or manipulating the number of iterations of a program loop or control-flow edge. This can have severe consequences depending on the context and is more challenging to detect. Furthermore, a stronger $\mathcal{ADV}$ can modify program code in memory at runtime through *physical access* without mounting sophisticated invasive physical attacks. $\mathcal{ADV}$ can replace the benign code memory with malicious code memory at runtime especially if the program code resides in an external off-chip memory [10] (**A5**).

Assumptions. We assume that $\mathcal{MOD}$ has access to the source and binary code of the target program and that static root of trust is in place by deploying conventional static (*binary*) attestation at load-time. This assures that $\mathcal{DEV}$ is executing unmodified program code, thus effectively mitigating attacks **A1**. Static attestation is a standard established mechanism assumed to be commonly deployed in embedded systems, while incurring no overhead on the application run time. Access to the source code is assumed for $\mathcal{MOD}$ to generate the control-flow graph of the target program and define the set of policies to be enforced/attested (described in more detail in III-A). Code injection attacks **A2** can be effectively prevented by marking memory as either writable or executable (W$\oplus$X). This mechanism, known as *Data Execution Prevention* (DEP) [13], is long-established and considered a standard assumption for these systems. Finally, we assume that expensive invasive physical attacks are out of scope (except attacks **A5** because they are realistic in practice). Thus, $\mathcal{ADV}$ cannot compromise hardware-protected memory used exclusively by CHASE that is not mapped to the software-accessible address space.

B. Requirements Analysis

To address the above, we derive the following requirements:

R1 *Configurable security:* Different security services with varying security guarantees should be supported. These should be configured at runtime depending on the threat level presumed for the deployment and functionality requirements.

R2 *Runtime security:* The security services should be capable of detecting different classes of runtime attacks (**A3** - **A5**) when activated. For at least one of these services, the attacks should also be detected with a sufficiently low latency for an impending exploit to be prevented in time (not just after-the-fact detection).

R3 *Minimal performance impact:* All services supported should incur minimal performance overhead on $\mathcal{DEV}$. At least one of these services should deterministically guarantee zero performance overhead and no impact on the application run time.

R4 *Accuracy & completeness:* All services, when enabled, should accurately capture, record and enforce or attest every control-flow event in the execution. No control-flow events can be dropped or bypass the activated security mechanism.

R5 *Secure communication:* Whenever applicable, attestation results should be securely reported to $\mathcal{MOD}$; they should be integrity-protected and fresh.

R6 *Reasonable logic overhead:* The hardware extension providing these security services should incur minimal memory and logic overhead to the baseline processor.

III. CHASE: High-Level Design

In light of the requirements described in § II, we present CHASE, the first consolidated hardware-assisted security extension that is *configurable by design* in the face of different attack classes § II-A based on their degree of difficulty to launch and (effectively) detect/mitigate. With this in mind, we describe the different security configuration modes supported in CHASE to mitigate these attacks and how they can be configured at runtime in § III-A. These modes aim to provide varying security guarantees by means of different security mechanisms to thwart different adversarial capabilities at different performance overhead costs. In § III-B, we describe the modular design of CHASE shown in Figure 2 and how its components can be configured to realize the different configuration modes at runtime.

Attack Categorization. As described in § II-A, attacks **A1** and **A2** are the most trivial to launch and mitigate. The former are mitigated by deploying static attestation at load-time. The latter are mitigated by deploying Data Execution Prevention (DEP), a long-established mechanism that does not affect application run time. Attacks **A3**, **A4**, and **A5** are the more challenging to launch and mitigate, and are currently the more sophisticated threats targeting embedded systems. CHASE provides different modes of configuration for different security guarantees against these attacks, thus fulfilling **R1** in § II-B. This enables the configuration of different security mechanisms at runtime while having the targeted application, functionality requirements and the threat level in mind, thus calibrating the performance/security trade-off flexibly. For real-time applications, for instance, particular configurations can be selected that do not affect the application run time, while providing an adequate level of security. For more vulnerable or less timing-critical applications, higher security guarantees can be provided by enabling other mechanisms, but at a higher performance overhead. This renders CHASE suitable for deployment in a wide spectrum of embedded systems with different use cases, while also taking into account the strict functionality requirements of timing-critical systems. We describe next how these different security services can be activated at runtime.

A. Security Configurations Scheme

Four configuration modes are supported by CHASE:

C1 On-device control flow verification

C2 On-device control flow enforcement

C3 Moderator-verified control flow attestation

C4 Moderator-verified executed instructions attestation

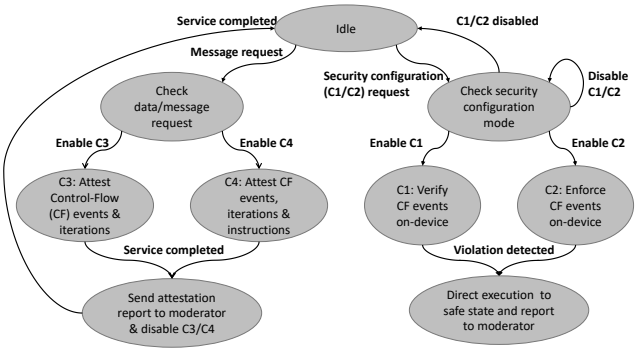


Fig. 1: Runtime activation of the CHASE configuration modes.

To enable these security mechanisms, a set of reference control-flow policies is generated and provisioned in a dedicated addressable policy memory on $\mathcal{DEV}$, while more complex policies are made available at $\mathcal{MOD}$. These policies are generated by means of offline one-time static and dynamic code analysis. The addresses used to access the policy memory for fetching the policies for indirect branch instructions are instrumented directly after the corresponding indirect jump instructions in the application binary (or source). Moreover, code analysis is used to generate a priori the list of security-critical data/messages that can be requested from $\mathcal{DEV}$, and the

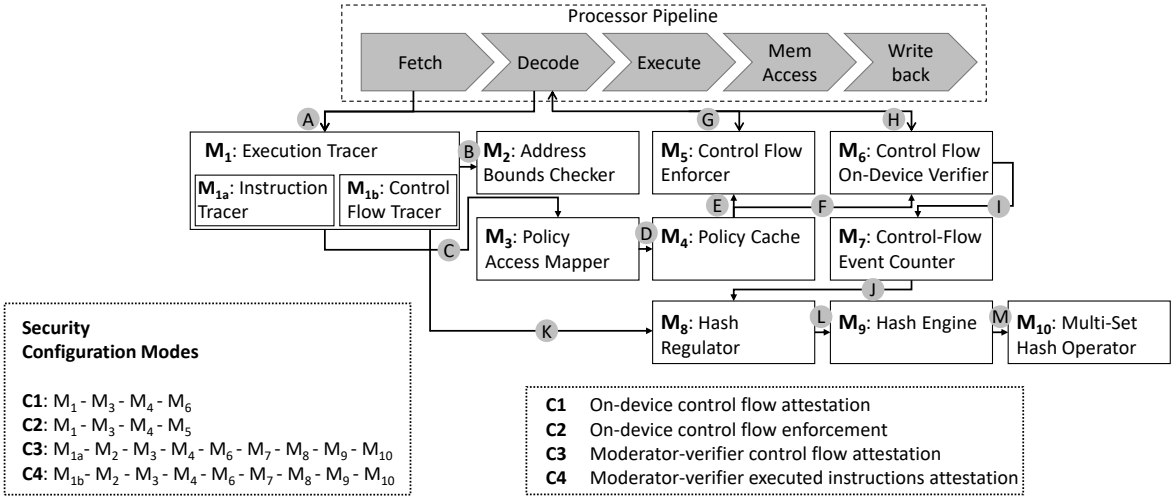


Fig. 2: CHASE high-level hardware architecture representing the modules activated for different security configuration modes.

corresponding software modules that contribute to the generation of the pertinent data/message, and their memory address bounds. This list is provisioned in a dedicated memory on $\mathcal{DEV}$ along with the data/message ID, and is used to restrict the attestation in modes **C3** and **C4** to the relevant software modules only, when a particular data/message is requested as described below.

Modes **C1** and **C2** can be enabled or disabled at any point during execution as shown in Figure 1, and are not bound to the execution of a particular data/message request. They verify or enforce, respectively, that the control flow of execution is valid so long as they are enabled. Modes **C3** and **C4**, on the other hand, are bound to a particular data/message request. If either mode is requested along with the data/message request, the attestation service continues to run until the request is completed. We describe next how the hardware modules of CHASE shown in Figure 2 are configured at runtime to realize the different configuration modes.

B. Security Configuration Modes

C1. Control-flow transfers are captured at runtime by module $\mathbf{M}_{1b}$ in Figure 2 while being executed. The captured control-flow events are checked against control-flow policies, which enlist the allowed destination addresses for every indirect branch instruction. These policies are in a dedicated policy memory and are pre-fetched into a dedicated on-chip cache (policy cache). The required policy address is mapped to the corresponding cache entry by module $\mathbf{M}_3$ and used to fetch the policies from cache in $\mathbf{M}_4$. $\mathbf{M}_6$ verifies the captured control-flow transfer by comparing it against the fetched policy if it is a forward transfer, e.g, a function call. The call site address is then pushed to the call-return matching stack in $\mathbf{M}_6$ if this is a `jump-and-link-register` instruction (function call). Backward control-flow transfers, e.g., returning to a call site, are verified by comparing the return address with a pre-defined number of most-recently call sites pushed onto the stack. The matching call site address and addresses pushed on top of it afterwards are all popped off the stack. The activated paths for this mode are A, C, D, F, and H through the activated modules $\mathbf{M}_{1b}$, $\mathbf{M}_3$, $\mathbf{M}_4$, and $\mathbf{M}_6$ as shown in Figure 2. This mode guarantees low-latency (within a few clock cycles) detection of illegal control-flow transfers and graceful mitigation of an impending exploit (**A3** attacks) while incurring zero performance overhead on the application run time (with caveats that we discuss in § V-B and § VI).

C2. A control-flow transfer is *not permitted* to occur unless it is validated on-device by $\mathbf{M}_5$ in this mode. This invasive enforcement of control-flow transfers guarantees prevention of executing illegal control-flow transfers (**A3** attacks) that violate the provisioned policies, while incurring minimal performance overhead. This overhead is variable and proportional to the number of control-flow transfers and the number of policy rules verified for each control-flow transfer. The activated paths for this mode are A, C, D, E, and G through the activated modules $\mathbf{M}_{1b}$, $\mathbf{M}_3$, $\mathbf{M}_4$, and $\mathbf{M}_5$.

C3. To detect more sophisticated control-flow and non-control-data attacks (**A3** and **A4** attacks), a more detailed attestation is performed that cannot be verified on-device owing to the complexity of the required code analysis and derived policies. Since this mode is activated for a specific data/message request, the current instruction address is checked by $\mathbf{M}_2$ to validate whether it is within the memory bounds of relevant program modules that should be attested for this particular data/message. This avoids attesting modules that may execute in parallel but are irrelevant to the requested data/message. If a control-flow event is within bounds, i.e., to be attested, its policy address is fetched from cache by $\mathbf{M}_3$ and $\mathbf{M}_4$ (similar to **C1** and **C2**). Then the control-flow event is verified on-device by $\mathbf{M}_6$ and the counter for this particular edge is incremented by $\mathbf{M}_7$ to keep track of the number of times a control-flow edge executes. Each executed edge is measured into a cryptographic hash computation by the hash engine ($\mathbf{M}_9$) and forwarded to the multi-set hash operator ($\mathbf{M}_{10}$). Both are regulated by ($\mathbf{M}_8$). The multi-set hash (MSH) function enables the computation of a single fixed-length hash digest for a set of elements (control-flow edges in this case) while allowing its members to occur multiple times, in which the order of the items does not affect the final value [14]. The MSH functionality significantly reduces the amount of information that needs to be sent by $\mathcal{DEV}$ to enable reconstructing the hash values at $\mathcal{MOD}$ side. The attestation report is assembled of a bitmap of executed control-flow edges and their iteration numbers as well as a single MSH measurement of the executed edges. The resulting attestation report is then sent to $\mathcal{MOD}$. $\mathcal{MOD}$ uses the bitmap to identify the executed edges and the number of times each edge was executed to identify benign behavior by the

expected control-flow edge iterations. Therefore, it can detect more sophisticated attacks which often involve a loop iteration counter getting maliciously compromised, thus causing the loop to execute for an abnormally different number of times than expected. $\mathcal{MOD}$ is assumed to be more computationally resourceful than $\mathcal{DEV}$ and provisioned with more complex fine-grained policies and heuristics in order to detect more sophisticated non-control-data attacks (**A4** attacks). The activated paths for this mode are (A), (B), (C), (D), (F), (H), (I), (J), (K), (L), and (M) through the activated modules $\mathbf{M}_{1b}$, $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$, $\mathbf{M}_6$, $\mathbf{M}_7$, $\mathbf{M}_8$, $\mathbf{M}_9$, and $\mathbf{M}_{10}$.

C4. This configuration mode assumes a much stronger adversary capable of manipulating the actual instructions that are executed at runtime without compromising control flow. To detect such an attack, every instruction executed, besides control flow, is captured by both $\mathbf{M}_{1a}$ and $\mathbf{M}_{1b}$ respectively, and included into the cryptographic hash computations. As in **C3** mode, a final MSH measurement is sent along with a bitmap of executed control-flow edges and their iteration numbers to $\mathcal{MOD}$ in order to detect attacks **A3**, **A4** and **A5**. The activated paths for this mode are ((A), (B), (C), (D), (F), (H), (I), (J), (K), (L), and (M)) through the activated modules $\mathbf{M}_{1a}$, $\mathbf{M}_{1b}$, $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$, $\mathbf{M}_6$, $\mathbf{M}_7$, $\mathbf{M}_8$, $\mathbf{M}_9$, and $\mathbf{M}_{10}$.

While providing progressively stronger and more sophisticated detection guarantees, configuration modes **C3** and **C4** come with significant latency that includes network communication with the remote moderator. On the other hand, modes **C1** and **C2** are performed on-device, thus incurring minimal (if any) clock cycle-level latency for detection/enforcement.

IV. CHASE: Hardware Architecture

We describe next the functionality of the hardware modules of CHASE used to enable the different configuration modes.

`Execution Tracer` $\mathbf{M}_1$ is tightly coupled to the processor pipeline. It tracks the execution flow of a software module/program by extracting several signals relevant to the execution from the processor pipeline at every clock cycle. It consists of two sub-modules: the Instruction Tracer which captures every instruction executed when mode **C4** is activated, and the Control-Flow Tracer which captures all signals relevant to control-flow transfers, as shown in Figure 2. This module requires tight interfacing with the Fetch and Decode stages of the pipeline to extract the program counter (PC), the instruction itself, and whether it is an indirect branch. Only indirect branches are exploitable by runtime attacks, and thus only these are captured and their destination addresses are extracted from the Decode stage.

`Bounds Checker` $\mathbf{M}_2$ controls which control-flow events are tracked and attested when either mode **C3** or **C4** is activated. For modes **C3** or **C4** only software modules that contribute to the generation of the requested data/message are attested during their execution (§ III-A). Therefore, the `Bounds Checker` compares the current PC with the address bounds of the relevant software modules. This requires two clock cycles and is interleaved with the operation of other CHASE modules for minimal performance overhead.

`Policy Access Mapper` $\mathbf{M}_3$ receives an address pointer to the policy memory from the `Execution Tracer` and maps it to corresponding cache entry in the `Policy Cache`, where the requested policy would be available. If the policy is already cached then the `Policy Access Mapper` fetches it from the `Policy Cache`. Otherwise, it issues a cache miss. The `Policy Access Mapper` also enables successive accesses to the `Policy Cache` in case more than one cache entry is required to store the policies for the pertinent control-flow transfer. The policies consist of the allowed source–destination address tuples and are organized in the policy memory per control-flow transfer, i.e., by source address, to achieve spatial locality in the `Policy Cache`, thus lower miss rates. As explained in § III-A, the policy memory addresses are instrumented within the program binary immediately after their corresponding indirect branch instructions, such that they get fetched anyway by the processor then invalidated (incurring no additional overhead). The `Execution Tracer` extracts the instrumented policy memory address before it gets invalidated.

`Policy Cache` $\mathbf{M}_4$ is accessed to fetch the cache line with the requested policy once the policy memory address is resolved by the `Policy Access Mapper`. The policy is then forwarded to either the `Control-Flow Verifier` or `Control-Flow Enforcer` depending on whether **C1** or **C2** is enabled respectively.

`Control-Flow Enforcer` $\mathbf{M}_5$ is tightly coupled with the processor Decode stage. For function calls and forward edges, the `Control-Flow Enforcer` ensures that the computed destination address matches one of the possible destination addresses in the relevant policy. For function returns or backward edges, the `Control-Flow Enforcer` utilizes the call-return matching stack to enforce returning to one of the call sites on the stack. Otherwise, it issues an interrupt signal to flush the pipeline and jump to an interrupt routine, which requires invasive integration with the pipeline.

`Control-Flow Verifier` $\mathbf{M}_6$ checks whether the computed control-flow destination address adheres to the pertinent policy by comparing it with the allowed destination addresses in parallel. If no match is found, then a control-flow violation is detected and communicated to $\mathcal{MOD}$. Execution may be gracefully interrupted and redirected to an application-dependent safe state. In case **C3** or **C4** is also enabled, the executed control-flow event is recorded in the metadata and included in the hash computation to report to $\mathcal{MOD}$. $\mathcal{MOD}$ can analyze the detected violation to confirm that it is indeed a violation and not a false positive. Otherwise, it can update the provisioned policies if necessary. The `Control-Flow Verifier` also informs the `Control-Flow (CF) Event Counter` with the executed control-flow event for further processing.

`Control-Flow (CF) Event Counter` $\mathbf{M}_7$ is only enabled for modes **C3** and **C4** and receives information from the `Control-Flow Verifier` on the executed control-flow transfers. The `CF Event Counter` generates and maintains a bitmap per software module that indicates which indirect edges and how many times they are executed by means of maintaining edge counters. It also maintains a list of control-flow transfers that were flagged as violating. At the end of the attestation epoch of **C3** or **C4**, the `CF Event Counter` outputs the metadata to be sent to $\mathcal{MOD}$ namely, the bitmaps, violated control-flow events and edge-counters.

`Hash Regulator`, **`Hash Engine`** and **`MSH Operator`** are only enabled in the two modes **C3** and **C4**. **`Hash Regulator`** $\mathbf{M}_8$ regulates the operation of the `Hash Engine` for different configuration modes and instructs it when to initialize/finalize a hash computation. When **C3** is enabled, it instructs the `Hash Engine` to compute a hash measurement over the executed control-flow event. It also guarantees that an executed control-flow transfer is hashed only once when it is encountered for the first time, thus avoiding exhausting the `Hash Engine` (with respect to power computation and clock cycles) in repeatedly computing the same hash values. When **C4** is enabled, it also forwards the executed instructions between consecutive control-flow events to the `Hash Engine` for hashing. Finally, it also regulates the forwarding of the computed hash values from the `Hash Engine` to the `MSH Operator`.

`Hash Engine` $\mathbf{M}_9$ is required to be a high-throughput and collision-resistant cryptographic hash function (**R5** in § II-B). We de-

ploy BLAKE2 [4] as the underlying hash function for the computation of the multi-set hash value [14] that will be reported to $\mathcal{MOD}$. More details specific to our BLAKE2 instantiation are presented in § V.

`MSH Operator` $\mathbf{M}_{10}$ receives the computed hash values from the `Hash Engine` to continuously generate the intermediate and final multi-set hash (MSH) values. The `MSH Operator` performs additive multi-set hashing using simple arithmetic operations ($+$ and mod) [14]. At the end of the attestation epoch, the final MSH-hash value is sent along with metadata to $\mathcal{MOD}$ for further inspection. The execution of the `Hash Engine` and `MSH Operator` are interleaved to achieve minimal performance overhead.

V. Implementation and Evaluation

We prototype CHASE by extending the open-source RISC-V Pulpino [5] on a Zedboard Zynq evaluation board. Hardware modules were implemented in Verilog and integrated with the processor, along with integrating modifications to the processor pipeline for capturing the required execution signals and enabling control-flow enforcement. In the following, we highlight the most crucial details relevant to our proof-of-concept (PoC) implementation and evaluation.

A. Hardware Prototyping

`Policy Access Mapper & Policy Cache.` In our PoC, we deploy a direct-mapped `Policy Cache` and a simple mod function for the `Policy Access Mapper` that requires a single clock cycle. The policy is fetched from the mapped cache entry in the `Policy Cache`. Similar to conventional processor caches, the cache organization, mapping and replacement policies are design decisions and can be applied differently for the `Policy Cache`. Furthermore, cache misses in the `Policy Cache` would correspond to misses in the instruction cache. In case several cache lines need to be accessed for fetching the policies of a control-flow transfer, successive accesses to the `Policy Cache` are pipelined with the operation of the `Control-Flow Verifier` and `Control-Flow Enforcer` to maintain minimal performance overhead. In our PoC, we assume a 64KB cache and a 64B cache line, such that a tuple of 16 32-bit addresses (thus up to 16 policies for one source address) correspond to one cache line. We synthesize the cache using 8 instances of 64Kb Block RAMs (BRAMs), such that the complete cache line (storing 16 policies) is fetched in one cycle.

`Hash Engine & Hash Regulator.` Blake2 is deployed as the underlying cryptographic hash function for the computation of the multi-set hash [14]. We utilize the Blake2s version, which consists of 10 rounds of computation. In each round, the compression function is applied in parallel to the columns of the internal state and then in parallel to its diagonals. We build our `Hash Engine` implementation on top of the open-source Verilog implementation of Blake2s [6]. The `Hash Regulator` is a simple controller that regulates the operation of `Hash Engine` for different configuration modes and instructs it when to initialize or finalize a hash computation. In **C4**, instructions executed after a destination address of a control-flow transfer and the source address of the next control-flow transfer are split into blocks of 512 bits such that the blocks are hashed individually by `Hash Engine`. These values are forwarded continuously to the `MSH Operator` for the final MSH-hash value computations.

[4]https://blake2.net/

[5]https://github.com/pulp-platform/pulpino

[6]https://github.com/secworks/blake2s

B. Area & Performance Overhead

Area. In Table I, we show a breakdown of the area overhead of the CHASE modules. CHASE consumes $7,556$ lookup-tables (LUTs) and $5,040$ Registers/Flip Flops (FFs), approximately 50% of the baseline Pulpino logic (**R6** in § II-B). However, the `Hash Engine` incurs 50% of this overhead and requires at least 24 cycles to finalize a hash computation. The rounds of the `Hash Engine` implementation can be unrolled to achieve a significantly higher throughput (at the cost of increased area overhead).

Performance. We evaluate sample control-flow exploits (ROP and simple JOP attacks) with CHASE, and show that the detection latency of an illegal control-flow transfer is at least 3 clock cycles after an indirect branch is decoded. For instance, the total number of instructions in the Pixhawk[7] firmware, an open-source flight controller for drones (a timing-critical application), is $324,996$ instructions including $6,210$ indirect branches. For **C2**, this incurs a performance overhead of 6% on the application run time, assuming that all the policies are cached and only one cache line (maximum of 16 policies) is required per branch instruction. For **C1**, this latency does not affect the application run time (a requirement for such timing-critical applications), since the control-flow transfer is verified after its execution. However, this only holds as long as there are at least 3 clock cycles between consecutive indirect branch instructions, which is satisfied for this particular benchmark. For other cases where this is not true, only certain indirect branch instructions can be verified while others are discarded, or the binary can be re-instrumented accordingly. Nevertheless, the low detection latency guarantees that an impending exploit can be prevented in time (**R3**).

TABLE I: Breakdown of CHASE Area Overhead

	LUTs	FFs	Memory
`Execution Tracer`	210	128	1KB
`Bounds Checker`	630	1,071	4 KB
`Policy Access Mapper`	7	2	-
`Policy Cache`	-	4	64 KB
`Control-Flow Verifier`	230	140	-
`Control-Flow Enforcer`	1053	287	-
`Control-Flow Event Counter`	240	288	34 KB
`MSH Operator`	384	256	-
`Hash Regulator` & `Hash Engine`	4,802	2,864	-

VI. Security Analysis

CHASE aims to provide a configurable defense and cover the attack space described in § II-A. To achieve this, CHASE is required to provide *accurate*, *authentic* and *low-latency* enforcement or on-device verification of control-flow transfers for modes **C1** or **C2**. For **C3** and **C4**, CHASE is required to provide *accurate*, *complete*, *authentic*, and *fresh* attestation of control flow (as well as executed instructions in **C4**) of the program running on $\mathcal{DEV}$.

Attestation and Network Attacks. In modes **C3** and **C4**, the recorded control-flow events as well as the executed instructions (for **C4**) are measured into a compact cryptographically-secure additive multi-set hash (MSH) digest $\texttt{MSH} = \big(\texttt{hash}(r) + \Sigma \texttt{Iter}_\texttt{i}.\texttt{hash}(\texttt{CFlow}_\texttt{i})\big) \bmod 2^\texttt{n}$, where r is a nonce, $\texttt{CFlow}_\texttt{i}$ is an executed control-flow edge (source-destination address pair), $\texttt{Iter}_\texttt{i}$ is the number of iterations of that edge, n is the bit length of $\texttt{CFlow}_\texttt{i}$ and $\texttt{hash}$ is the underlying hash function used (Blake2 in CHASE). In order to evade detection of control-flow/code manipulation attacks, $\mathcal{ADV}$ is required to find a sequence of control-flow events/instructions (another multi-set) that maps to the the same MSH

[7]http://pixhawk.org/

value. However, this is not feasible since the chosen additive MSH is multiset-collision resistant where the hardness of finding collisions is reduced to the hardness of breaking the underlying hash function, which is the second pre-image resistant Blake2 in our design (**R5**).

Moreover, every attestation report is authenticated by $\mathcal{DEV}$ along with a monotonic counter `ctr` using a cryptographically-secure digital signature $\sigma = \texttt{sign}\{sk_{\mathcal{DEV}}; \texttt{bitmap}_{\texttt{CF}} \| \texttt{Iter}_i \| r \| \texttt{MSH} \| \texttt{ctr}\}$ based on $\mathcal{DEV}$'s signing key $sk_{\mathcal{DEV}}$. $sk_{\mathcal{DEV}}$ is stored in hardware-protected memory that is only accessible by CHASE. The signature and secure storage of the key guarantee the *authenticity* of the report while the monotonic counter ensures its *freshness* (**R5**). Note that, the monotonic counter is backed in a non-volatile memory and is non-resettable even when $\mathcal{DEV}$ is reset. Finally, since attestation is coupled with program execution at $\mathcal{DEV}$, Time-of-Check-Time-of-Use (TOCTOU) attacks on attestation are prevented.

Malware and Code Injection Attacks. Recall that the adversary is incapable of modifying the software binary at load-time (malware injection) as well as the control-flow policies which are fetched from external memory into on-chip memory. This is due to static attestation which allows the detection of such tampering, thus mitigating attacks **A1**. Furthermore, code injection attacks are effectively prevented through DEP (W⊕X)), thus mitigating attacks **A2**.

Runtime Code-Reuse Attacks. CHASE uses hardware modules that are tightly integrated with the processor to extract the control-flow information and executed instructions directly from the processor pipeline. This guarantees that all control-flow events and executed instructions are recorded. CHASE hardware also guarantees that all control-flow transfers are verified against their respective policies, thus, ensuring that the provisioned control-flow policies are accurately and completely enforced. The security guarantees with respect to the detection of runtime attacks are as good as the provisioned control-flow policies, and the code analysis that generated the policies. So long as any configuration modes in CHASE is enabled, at least every executed control-flow transfer is directly captured from the processor and either verified or enforced (for modes **C1** and **C2** respectively) using the provisioned control-flow policies or measured (for modes **C3** and **C4**). Thus, in order to ensure that no control-flow transfers are dropped, `Execution Tracer` of CHASE is padded with a First-In-First-Out (FIFO) structure that buffers incoming control-flow transfers. This is required for the unlikely event that multiple indirect jump instructions execute consecutively (**R1** and **R4**). While CHASE ensures that all buffered control-flow transfers are verified with their respective policies, the detection latency of potential violations is increased. Nevertheless, software developers are advised not to program multiple indirect branches consecutively.

This accurate tracking and enforcement/verification of control-flow edges in CHASE guarantees the detection of explicit control-flow hijacking attacks (**A3**). The call-return matching stack also provides additional guarantees on context-sensitive enforcement/verification of backward edges, i.e., returning to correct call sites. Moreover in **C3** and **C4**, tracking the number of times each edge iterates and sending the respective bitmap to $\mathcal{MOD}$ enables the detection of all data-oriented attacks that do not directly hijack the control flow but maliciously compromise the expected number of loop iterations (attacks **A4**). In **C4**, tracking and measuring every instruction executed (not only control flow) allows $\mathcal{MOD}$ to detect runtime code manipulation attacks (**A2** and **A5**). This assumes that $\mathcal{MOD}$ has knowledge of the program source code and the benign number of loop iterations for a given service/message by means of code analysis (see § II-A) (**R2**).

Finally, since CHASE is hardware-based, it cannot be compromised by malicious software. Moreover, all on-chip cache/memory utilized by CHASE, e.g., for policies, is hardware-protected and not mapped to software-accessible address space, and hence protected from remote software attacks.

Figure 3 shows an example of a control-flow violation at runtime. The solid arrows represent the expected benign execution path, while the dashed arrows respresent the violating control-flow edges, assuming the function pointer in the writable data memory was overwritten by the attacker. Assuming **C3** is enabled, the attestation report will differ from the reference in the bitmap of the executed edges, iterations per edge as well as the final MSH value computed.

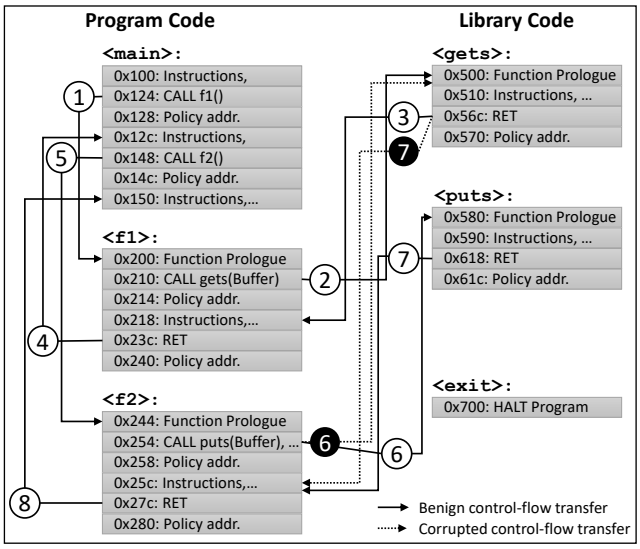


Fig. 3: Detection of a control-flow violation by CHASE. **Expected control-flow path (benign)**: ① (0x124, 0x200), ② (0x210, 0x500), ③ (0x56c, 0x218), ④ (0x23c, 0x12c), ⑤ (0x148, 0x244), ⑥ (0x254, 0x580), ⑦ (0x618, 0x25c), ⑧ (27c, 0x150), etc. **Traced control-flow path (corrupted)**: ① (0x124, 0x200), ② (0x210, 0x500), ③ (0x56c, 0x218), ④ (0x23c, 0x12c), ⑤ (0x148, 0x244), ❻ (0x254, 0x500), ❼ (0x56c, 0x25c), ⑧ (27c, 0x150), etc.

Physical Attacks. Expensive invasive/semi-invasive physical attacks are out of scope in this work, thus CHASE hardware is assumed secure against such attacks that would compromise its accuracy/functionality (**R4**). However, other realistic physical attacks that manipulate the program code at runtime, as well as fault injection attacks are captured by CHASE and detected by $\mathcal{MOD}$ in the mode **C4** since CHASE captures all executed instructions.

VII. Related Work

Static Attestation. Attestation aims at enabling a trusted third party to check the trustworthiness of the software on another device. Approaches to static attestation include: (1) Software-based attestation [15], [16] that allows the attestation of legacy and low-end computing devices while requiring no secure hardware but relying on strong assumptions, and thus have been attacked [17], (2) Hardware-based attestation [18], [19] that requires complex and/or security hardware (e.g., trusted platform module – TPM), and (3) Hybrid attestation [20]–[22] that is based on hardware/software co-design aiming at reducing the hardware security required for remote attestation. Static attestation, however, cannot detect runtime attacks.

Runtime Integrity. Many defenses have been proposed in recent years to mitigate runtime exploits [1]. Control-Flow Integrity

(CFI) [3] ensures that a program follows a valid path in its control-flow graph (CFG). However, CFI does not mitigate non-control-data and DOP attacks. Code randomization [23] randomizes the code layout, but a branch instruction can still be exploited to jump to the target address of choice. Code-Pointer Integrity (CPI) [24] aims at ensuring the integrity of code pointers but also does not mitigate non-control-data attacks. Defenses has been proposed for mitigating pure data-oriented attacks such as data-flow integrity enforcement/isolation [12], [25], [26], but they all incur a non-negligible performance overhead.

Runtime Attestation. Control-flow attestation was proposed in [4] and aimed to allow the verifier to attest the measure and record the control-flow path executed on the prover. However, code instrumentation was required and prohibitively high performance overhead was incurred on the prover. Subsequent schemes have been proposed [4], [9]–[11] to leverage hardware assistance for recording runtime execution events in parallel to program execution, and without code instrumentation, thus reducing the overhead on the prover significantly and tackling stronger adversarial capabilities [10], [11]. However, existing schemes each tackle different adversarial capabilities with no scheme providing a consolidated or configurable defense.

Defenses for Real-Time Systems. Some defenses have been recently proposed for providing integrity to real-time systems and the requirements of applying remote attestation to safety-critical systems was investigated recently in [27]. SeED [28] enables non-interactive attestation, while ERASMUS [29] proposes periodic self-measurements of the prover's software that are occasionally reported to a remote verifier, thus providing applicability to real-time systems, but only providing static integrity. DIAT [30] proposes data-integrity attestation for collaborating real-time systems but still incurs significant performance overhead to the application run time. ECFI [5] proposes a CFI mechanism for PLCs which gives priority to the PLC's runtime operation, yet it still incurs a performance overhead of up to 8.3%. Existing solutions cannot provide runtime security guarantees for a real-time system without incurring a performance overhead on the application, unlike CHASE.

VIII. Conclusion

In this work, we presented the first hardware-assisted security extension CHASE, that consolidates different modular defenses that can be configured at runtime to mitigate different adversarial capabilities, thus effectively covering a larger attack space than existing defenses. This enables the calibration of the security/performance trade-off by selecting the desired level of security and thus the corresponding performance overhead. Moreover, CHASE also provides a non-intrusive control-flow verification mechanism that does not affect the application run time, yet detects violations with minimal latency, making it applicable to timing-critical systems.

Acknowledgments. This work is supported by the German Research Foundation (DFG) within CRC 1119 CROSSING and HWSec, by the German Federal Ministry of Education and Research (BMBF) within CRISP, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] L. Szekeres et al., "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[2] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *ACM CCS*, 2007.

[3] M. Abadi et al., "Control-flow integrity: Principles, implementations, and applications," *ACM TISSEC*, vol. 13, no. 1, 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960

[4] T. Abera et al., "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[5] A. Abbasi et al., "ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2017.

[6] K. Stouffer et al., "Guide to Industrial Control Systems (ICS) Security," *NIST special publication*, vol. 800, no. 82, pp. 16–16, 2011.

[7] L. Davi et al., "HAFIX: Hardware-Assisted Flow Integrity eXtension," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[8] D. Sullivan et al., "Strategy without Tactics: Policy-Agnostic Hardware-Enhanced Control-Flow Integrity," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[9] G. Dessouky et al., "LO-FAT: Low-overhead control flow attestation in hardware," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017.

[10] S. Zeitouni et al., "ATRIUM: Runtime Attestation Resilient Under Memory Attacks," in *International Conference On Computer Aided Design (ICCAD)*, 2017.

[11] G. Dessouky et al., "LiteHAX: Lightweight Hardware-assisted Attestation of Program Execution," in *International Conference On Computer Aided Design (ICCAD)*, 2018.

[12] C. Song et al., "HDFI: Hardware-Assisted Data-Flow Isolation," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[13] Hewlett-Packard, "Data execution prevention," 2006.

[14] D. Clarke et al., "Incremental multiset hash functions and their application to memory integrity checking," in *International conference on the theory and application of cryptology and information security*. Springer, 2003, pp. 188–207.

[15] R. Gardner, S. Garera, and A. Rubin, "Detecting Code Alteration by Creating a Temporary Memory Bottleneck," *IEEE Transactions on Information Forensics and Security*, 2009.

[16] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the integrity of peripherals' firmware," in *ACM Conference on Computer and Communications Security*, 2011.

[17] G. Wurster, P. C. Van Oorschot, and A. Somayaji, "A Generic Attack on Checksumming-based Software Tamper Resistance," in *IEEE Symposium on Security and Privacy (Oakland)*, 2005.

[18] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot — A Coprocessor-based Kernel Runtime Integrity Monitor," in *USENIX Security Symposium*. USENIX Association, 2004.

[19] X. Kovah et al., "New Results for Timing-Based Attestation," in *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[20] K. Eldefrawy et al., "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," in *Network and Distributed System Security Symposium*, 2012.

[21] P. Koeberl et al., "TrustLite: A Security Architecture for Tiny Embedded Devices," in *European Conference on Computer Systems*, 2014.

[22] A. Francillon et al., "A minimalist approach to remote attestation," in *Design, Automation & Test in Europe*, 2014.

[23] P. Larsen et al., "SoK: Automated software diversity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[24] V. Kuznetsov et al., "Code-Pointer Integrity," in *USENIX Symposium on Operating Systems Design and Implementation (ODSI)*. USENIX Association, 2014.

[25] M. Castro et al., "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298470

[26] T. Nyman et al., "HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement," *CoRR*, vol. abs/1705.10295, 2017. [Online]. Available: http://arxiv.org/abs/1705.10295

[27] X. Carpent et al., "Invited: Reconciling Remote Attestation and Safety-Critical Operation on Simple IoT Devices," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2018.

[28] A. Ibrahim et al., "SeED: Secure Non-interactive Attestation for Embedded Devices," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[29] X. Carpent et al., "ERASMUS: Efficient Remote Attestation via Self-Measurement for Unattended Settings," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018.

[30] T. Abera et al., "DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous System," in *Annual Network & Distributed System Security Symposium (NDSS)*, 2019.

F

HYBCACHE: HYBRID SIDE-CHANNEL-RESILIENT CACHES FOR TRUSTED EXECUTION ENVIRONMENTS

[41] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In USENIX Security. USENIX Association, 2020. Core Rank A*.

HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments

Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi,
Technische Universität Darmstadt

https://www.usenix.org/conference/usenixsecurity20/presentation/dessouky

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

HYBCACHE: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments

Ghada Dessouky, Tommaso Frassetto, Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
{ghada.dessouky, tommaso.frassetto, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

Modern multi-core processors share cache resources for maximum cache utilization and performance gains. However, this leaves the cache vulnerable to side-channel attacks, where inherent timing differences in shared cache behavior are exploited to infer information on the victim's execution patterns, ultimately leaking private information such as a secret key. The root cause for these attacks is mutually distrusting processes sharing the cache entries and accessing them in a deterministic and consistent manner. Various defenses against cache side-channel attacks have been proposed. However, they suffer from serious shortcomings: they either degrade performance significantly, impose impractical restrictions, or can only defeat certain classes of these attacks. More importantly, they assume that side-channel-resilient caches are required for the entire execution workload and do not allow the possibility to selectively enable the mitigation only for the security-critical portion of the workload.

We present a generic mechanism for a flexible and soft partitioning of set-associative caches and propose a hybrid cache architecture, called HYBCACHE. HYBCACHE can be configured to selectively apply side-channel-resilient cache behavior only for isolated execution domains, while providing the non-isolated execution with conventional cache behavior, capacity and performance. An isolation domain can include one or more processes, specific portions of code, or a Trusted Execution Environment (e.g., SGX or TrustZone). We show that, with minimal hardware modifications and kernel support, HYBCACHE can provide side-channel-resilient cache *only* for isolated execution with a performance overhead of 3.5–5%, while incurring no performance overhead for the remaining execution workload. We provide a simulator-based and hardware implementation of HYBCACHE to evaluate the performance and area overheads, and show how HYBCACHE mitigates typical access-based and contention-based cache attacks.

1 Introduction

For decades now, upcoming processor generations are being augmented with novel performance-enhancing capabilities. Performance and security of processor architectures and microarchitectures are considered exclusively independent design metrics, with architects primarily focused on the more tangible performance benefits. However, the recent outbreak of micro-architectural cross-layer attacks [4–6, 18, 19, 22, 42, 44, 46, 47, 50, 56, 59, 68, 70, 79], has demonstrated the critical and long-ignored effects of micro-architectural performance optimizations on systems from a security standpoint. It is becoming evident how performance and security are at conflict with each other unless architects address the design trade-off early on and not as an afterthought.

One prominent performance feature and the subject of a wide range of recent architectural attacks is the use of caches and cache-like structures to provide orders-of-magnitude faster memory accesses. The intrinsic timing difference between a cache hit and miss is one of various *side channels* that can be exploited by an adversary process via a carefully crafted side-channel attack to infer the memory access patterns of a victim process [23, 25–29, 34, 35, 38, 54, 61, 71, 77, 78]. Consequently, the adversary can leak unauthorized information, such as a private key, hence violating the confidentiality and isolation of the victim process.

Cache Side-Channel Attacks. In earlier years, cache side-channel attacks have been shown to compromise cryptographic implementations [8, 54, 61, 78]. More recently, attack variants such as *Prime + Probe* [34, 38, 54, 61] and *Flush + Reload* attacks [29, 78] are being demonstrated on a much larger scale. They have been shown to bypass address space layout randomization (ASLR) [23, 25], infer keystroke behavior [26, 27], or leak privacy-sensitive human genome indexing computation [11], whereby millions of platforms using various architectures have been shown vulnerable to such attacks. The attacks require an adversary to orchestrate particular cache evictions of target memory addresses of interest and

after a time interval measure its own memory access latencies or observe relevant computation and profile how it has been affected. This enables the adversary to deduce the victim's memory access patterns and infer dependent secrets. Cache side-channel attacks have been shown to exploit core-specific caches as well as shared last-level caches across different cores or virtual machines [27,38,54]. Even hardware-security extensions and trusted execution environments (TEEs) such as Intel SGX [13, 33] and ARM TrustZone [7] are not immune to these attacks. While they do not claim cache side-channel security, recent cache side-channel attacks targeting SGX [11,21,60,66] and TrustZone [49,80] have been shown to compromise the acclaimed privacy and isolation guarantees of these security architectures, thus undermining their very purpose.

Existing Cache Defenses. To defeat cache side-channel attacks, there has been extensive research on techniques to identify and mitigate information leaks in a software's memory access patterns [16, 17, 45]. However, mitigating these leaks efficiently for arbitrary software (beyond cryptographic implementations) remains impractical and challenging. Alternatively, hardware-based and software approaches have been proposed to modify the cache organization itself to limit cache interference across different security domains. Examples include modifying replacement and leveraging inclusion policies [39,76], as well as approaches that rely on cache partitioning [24,40,41,51,72,73,82], and randomization/obfuscation-based schemes [52,53,63,69,73] to randomize the relation between the memory address and its cache set index.

While strict cache partitioning is the intuitive approach to provide complete cache isolation and non-interference between mutually distrusting processes, it remains highly impractical and prevents efficient cache utilization. On the other hand, randomization-based approaches make the attacks computationally much more difficult by randomizing the mapping of memory addresses to cache sets. However, existing schemes either require complex management logic, impose particular restrictions, rely on weak cryptographic functions, or mitigate only some classes of cache side-channel attacks. Most importantly, all of the aforementioned schemes are designed to provide side-channel cache protection for the entire code execution, which is actually not required in practice.

Our Goals. We observe that usually the majority of the code is not security-critical. Typically, a small portion of the code is security-critical and requires cache-based side-channel resilience. Moreover, this security-critical portion of the code is often already running in an isolated environment, such as in a TEE or in an isolated process. In these cases, a trusted component, namely the processor hardware or microcode or the operating system kernel, enforces this isolation. We aim to leverage and extend this existing isolation mechanism to also selectively enable side-channel resilience for the caches *only* for the portion of the code that needs it, without reducing the cache performance for the remaining non-isolated code. In doing so, we practically address the persistent performance-security trade-off of caches by providing the system administrator with a "tuning knob" to configure by balancing and isolating the workload as required. Consequently, s/he can tune the resulting cache side-channel resilience, utilization, and performance, while guaranteeing no performance overhead is incurred on the non-isolated portion of the code execution. Only the isolated (usually the minority) portion is subject to a reasonable reduction in cache capacity and performance – the cost of increased security guarantees.

To achieve this flexible and hybrid cache behavior, we introduce HYBCACHE, a generic mechanism that protects isolated code from cache side-channel attacks without reducing the cache performance for the remaining non-isolated code. In HYBCACHE, isolated execution only uses a pre-defined (small) number of cache ways[1] in each set of a set-associative cache. It uses these ways fully-associatively, while for eviction random victim cache lines are selected to be replaced by new ones, thus breaking the set-associativity and removing the root cause of access leakage. Non-isolated execution uses all cache ways set-associatively as usual, without any performance overhead. While isolated and non-isolated execution may compete for the use of some ways in the cache, the random replacement policy and fully-associative mapping used by the isolated execution prevent leaking information about the accessed memory locations (and their cache set mapping) to the non-isolated execution, thus making the precomputation and construction of an eviction set impossible. Moreover, HYBCACHE flexibly supports multiple, mutually distrusting isolated execution domains while preserving the above security guarantees individually for each domain.

HYBCACHE is architecture-agnostic, and can be seamlessly integrated with any isolation mechanism (TEEs or inter-process isolation); the definition of the isolation domains and the distribution of the workload is left up to the system administrator. HYBCACHE is backward compatible by design; it provides conventional set-associative caches for the workload if the side-channel resilience feature is not supported.

Contributions. The main contributions of this paper are as follows.

- We present HYBCACHE, the first cache architecture designed to provide flexible configuration of cache side-channel resilience by selectively enabling it for isolated execution without degrading the performance and available cache capacity of non-isolated execution.
- We evaluate the performance overhead of a simulator-based implementation of HYBCACHE and show that it is less than 5% for the SPEC2006 benchmarks suite,

[1] *Ways* are different available entries in a cache set to which a particular memory address can be allocated.

and estimate the memory and area overheads of a cycle-accurate hardware implementation of HYBCACHE.
- We show – through our security analysis – how breaking set-associative mapping and shared cache lines between mutually distrusting isolation domains (which are the root causes for typical cache side-channel attacks besides the intrinsic cache sharing and competition) mitigates typical contention-based and access-based cache attacks.

2 Cache Organization, Attacks and Defenses

We briefly present the typical cache organization, as well as recent cache side-channel attacks that are within the scope of our work, and limitations of existing defenses.

2.1 Cache Organization

Cache Structure. Caches are typically arranged in a hierarchy of fastest/closest/smallest to slowest/furthest/largest levels of cache, respectively L1, L2, and L3 cache/last-level-cache (LLC). Each core incorporates its L1 and L2 caches and shares the LLC with other on-chip cores. A cache consists of the storage of the actual cached data/instructions and the *tag* bits of their corresponding memory addresses. Cache memory is organized into fixed-size memory blocks, called *cache lines* each of size B bytes. Set-associative caches are organized into S sets of W ways each (called a W-way set-associative cache) where each way can be used to store a cache line. A single cache line can only be allocated to only one of the cache sets, but can occupy any of the ways within this cache set. The least significant $log_2\ B$ bits are the *block offset* bits that indicate which byte block within the B-Byte cache line is requested. The next $log_2\ S$ bits are the *index* bits used to locate the correct cache set. The remaining most significant bits are the *tag* bits for each cache line.

In a set-associative cache, once the cache set of a requested address is located, the *tag* bits of the address are matched against the tags of the cache lines in the set to identify if it is a cache hit. If no match is found, then it is a miss at this cache level, and the request is sent down to the next lower-level cache in the hierarchy until the requested cache line is found or fetched from main memory (cache miss). However, in a fully-associative cache, a cache line can be placed in any of the cache ways where the entire cache serves as one set. No index bits are required, but only $log_2\ B$ block offset bits and the rest of the bits serve as tag bits.

Eviction and Replacement. Due to set-associativity and limited cache capacity, cache contention and capacity misses occur where a cache line must be evicted in favor of the new cache line. Which cache line to evict depends on the replacement policy deployed, some of which include First-in-First-Out (FIFO), Least-Recently-Used (LRU), pseudo-LRU, Least-Frequently-Used (LFU), Not-Recently-Used (NRU), random and pseudo-random replacement policies. In practice, approximations to LRU (pseudo-LRU) and random replacement (pseudo-random) are usually deployed.

2.2 Cache Side-Channel Attacks

Cache side-channel attacks pose a critical threat to trusted computing and underlie more proliferating side-channel attacks such as the Spectre [44] and Meltdown [50] variants. Different classes of these attacks have been demonstrated on all platforms and architectures ranging from mobile and embedded devices [49] to server computing systems [34, 54, 81]. They have also been shown to undermine the isolation guarantees of trusted execution environments, like Intel SGX [11, 21, 60, 66] and ARM TrustZone [49, 80]. Such attacks have been shown to infer both fine-grained and coarse-grained private data and operations, such as bypassing address space layout randomization (ASLR) [23, 25], inferring keystroke behavior [26, 27], or leaking privacy-sensitive human genome indexing computation [11], as well as RSA [54, 81] and AES [10, 34] decryption keys.

Cache side-channel attacks exploit the inherent leakage resulting from the timing latency difference between cache hits and misses. This is then used to infer privacy/security-critical information about the victim's execution. In an offline phase, the attacker must first identify the target addresses of interest (by means of static and dynamic code analysis of the victim program) whose access patterns leak the desired information about the victim's execution, such as a private encryption key. In an online phase, the attacker measures the timing latency of its memory accesses or the victim's computation time to infer the desired information.

To demonstrate how a simple cache attack works, consider the pseudo-code of the Montgomery ladder implementation for the modular exponentiation algorithm shown in Algorithm 1. Modular exponentiation is the operation of raising a number b to the exponent e modulo m to compute $b^e\ mod\ m$ and is used in many encryption algorithms such as RSA. Leaking the exponent e may reveal the private key. As shown in Algorithm 1, the operations performed for each of the exponent bits directly correspond to the value of the bit. If the exponent bit is a zero, the instruction in Line 5 is executed. If the exponent bit is a one, the instruction in Line 9 is executed. An attacker that can observe or deduce these execution patterns can thus disclose the value of each corresponding exponent bit, and eventually recover the encryption key [78, 81]. S/he, however, needs to identify the target addresses that need to be observed (the addresses of the instructions in Lines 5 and 9 in this example) in the victim program and accordingly construct the eviction set. The eviction set is a collection of addresses that are mapped to the same specific cache set to which the target addresses are also mapped. The attacker uses this eviction set to evict the contents of the whole set in the cache, and therefore guarantee to successfully evict the target

addresses from the caches. Consequently, s/he measures the timing latency of its own memory accesses after a time interval to deduce whether the victim has accessed these target addresses.

Algorithm 1: Montgomery Ladder RSA Implementation

```
   Input: base b, modulo m, exponent e = (e_{n-1}...e_0)_2
   Output: b^e mod m
1  R_0 ← 1; R_1 ← b;
2  for i from n-1 downto 0 do
3      if e_i = 0 then
4          R_1 ← R_0 × R_1 mod m;
5          R_0 ← R_0 × R_0 mod m;
6      end
7      if e_i = 1 then
8          R_0 ← R_0 × R_1 mod m;
9          R_1 ← R_1 × R_1 mod m;
10     end
11 end
12 return R_0;
```

The online phase of these attacks consists of three main steps: *Eviction*, *Waiting* and *Analysis*. The attacker uses the eviction set to *evict* the victim's target addresses from the cache. Next, the attacker *waits* an interval of time to allow the victim to access the target addresses. Then the attacker measures and *analyzes* its access time measurements to determine if the victim has accessed the target addresses. This is repeated as many times as the attacker requires to collect sufficient traces to recover the exponent bits.

The different techniques used by the attacker to perform the eviction can be classified into two main approaches, either access-based or contention-based. In access-based attacks such as Flush + Reload [29, 78], Flush + Flush [26], Invalidate + Transfer [35], and Flush + Prefetch [25], the attacker accesses the target addresses directly by flushing them out of the cache using the dedicated *clflush* instruction [2] and possibly exploiting timing leakage from the execution of the *clflush* instruction [26]. This invalidates the lines containing these addresses and writes them back to memory. Evict + Reload [27] attacks have also been shown which do not require the *clflush* instruction, but instead evict specific cache sets by accessing physically congruent addresses. These attacks are only feasible in case of shared memory pages between the attacker and victim, usually in the form of shared libraries. Otherwise, an attacker resorts to contention-based attacks such as Prime + Probe [34, 38, 54, 61, 77], Prime + Abort [15], Evict + Time [23, 61], alias-driven attacks [28], and indirect Memory Management Unit (MMU)-based cache attacks [71], where s/he constructs an eviction set and uses it to trigger and exploit a cache contention in the same cache set as the target addresses, thus evicting cache lines containing the target addresses from the pertinent cache set.

The waiting interval should be selected and synchronized such that the victim is expected to access the target address at least once before the attacker analyzes the collected observations. By analyzing the collected observations, the attacker determines whether the target address was indeed accessed by the victim. This is achieved by different techniques depending on the attack approach, either the adversary measures the overall time needed by the victim process to perform certain computations [8, 10], or probes the cache with eviction sets and profiles cache activity to deduce which memory addresses were accessed [34, 38, 54, 77, 78], or accesses target memory addresses and measures the timing of these individual accesses [29, 61]. Alternatively, the adversary can also read values of addresses from the main memory to see whether cache lines that contain cacheable target addresses have been evicted to memory [28].

Cache-collision timing attacks exploit cache collisions that the victim experiences due to its cache utilization, e.g., after a sequence of lookups performed by a table-driven software implementation of an encryption scheme, such as AES [10]. These attacks are out of scope in this work since they are not common, are specific to certain software implementations, and can only be mitigated by adapting the implementation or locking the relevant cache lines after pre-loading them.

2.3 Limitations of Existing Defenses

To mitigate these attacks, software-based countermeasures and modified cache architectures have been proposed in recent years, which we cover in depth in the Related Work (Section 8). These can be classified into two main paradigms: 1) applying cache partitioning to provide strict isolation, or 2) applying randomization or noise to make the attacks computationally impractical. However, all proposed countermeasures to date either impact performance significantly, require explicit programmer's annotations, are not seamlessly compatible with existing software requirements such as the use of shared libraries, are architecture-specific, or do not defend against all classes of attacks. Most importantly, all existing defenses apply their side-channel cache protection for the entire execution workload.

In practice, cache side-channel resilience is only required for the security-critical (usually smaller) portion of the workload that is allocated to execute in isolation. Thus, non-isolated execution should not suffer any resulting performance costs. To address this in this work, we propose a modified hybrid cache microarchitecture that enables side-channel resilience only for the isolated portion of execution, while retaining the conventional cache behavior and performance for the non-isolated execution.

3 Adversary Model and Assumptions

To provide side-channel-resilient cache accesses for only security-critical isolated execution, we propose a hybrid *soft* partitioning scheme for set-associative memory structures.

In this work, we apply it to caches and call it HYBCACHE. HYBCACHE aims to provide cache-based side-channel resilience to the security-critical or privacy-sensitive workload that is allocated to one or more **I**solated Execution **D**omains (I-Domains), while maintaining conventional cache behavior for non-critical execution that is allocated to the **N**on-**I**solated Execution **D**omain (NI-Domain). HYBCACHE assumes an adversary capable of mounting the attacks described in Section 2.2 and is designed to mitigate them.

Furthermore, the construction of HYBCACHE is based on the following assumptions:

A1 Security-critical code that requires side-channel resilience is already allocated to an isolated component, like a process or a TEE (enclave).

A recent trend in the design of complex applications, like web browsers, is to compartmentalize them using multiple processes. As an example, all major browsers spawn a dedicated process for every tab [43] and some even use a dedicated process to better isolate privileged components [58]. Similarly, the widespread availability of TEEs, like SGX, encourages developers to encapsulate sensitive components of their code in protected environments.

A2 Isolated execution is the minority of the workload.

Isolation works best when the isolated component is as small as possible, thus reducing the attack surface. This complies with the intended usage of TEEs like SGX where only small sensitive components of the code would be allocated to the TEE. Hence, we assume only the minority of the workload needs to be isolated. HYBCACHE still provides the same security guarantees if the majority of the workload is isolated, but the performance of the isolated execution would suffer.

A3 Sensitive code only uses writable shared memory for I/O (if at all), and access patterns to this shared memory do not leak any information.

Isolated code should focus on processing some local data, while I/O needs should be limited to copying the input(s) into the isolated component, and copying the output(s) out of the component. Both of these procedures just access the data sequentially; thus, the access patterns during I/O do not depend on the data and does not leak any information.

A4 The attacker is not in the same I-Domain as the victim.

HYBCACHE is designed to isolate mutually distrusting I-Domains and thus, we must assume the attacker and the victim are not in the same I-Domain. Note that, as a consequence of A3, if a process handles sensitive data and has multiple threads, they must all be in the same I-Domain, since they share the entire address space. In cases where isolation between threads sharing the same address space is also required, HYBCACHE can, in principle, provide intra-process isolation as discussed later in Section 7.

4 Hybrid Cache (HYBCACHE)

We systematically analyzed existing contention-based and access-based cache attacks in the literature (Section 2.2) to identify their common root causes (besides the intrinsic sharing of cache entries and latency difference between a cache hit and miss). Cache side-channel attacks are, by nature, very specific to the victim program and may exploit attack-specific features such as the side-channel leakage of the *clflush* [26] or prefetch instructions [25]. Nevertheless, each one of these attacks is primarily caused by one or both of the following root causes: shared memory pages (and cache lines) between mutually distrusting code, and deterministic and fixed set-associativity of cache structures, which enables targeted cache set contention by pre-computed eviction sets.

4.1 Requirements Derivation

In light of the above, HYBCACHE should provide side-channel resilience between different isolation domains with respect to their cache utilization. An adversary process sharing the cache with a victim process should not be able to distinguish which memory locations a victim accesses. Nevertheless, we emphasize that the only approach to enforce complete non-interference between different domains is by strict static cache partitioning, such that no cache resources are shared, and thus zero information leakage occurs. On the other hand, this is impractical, and results in inefficient cache utilization from a performance standpoint. Our key objective in this work is to practically address and accommodate this persistent performance/security trade-off of cache structures by providing sufficiently strong cache side-channel-resilience, such that practical and typical cache side-channel attacks become effectively infeasible without necessarily enforcing complete non-interference. Additionally, we desire that this security guarantee is run-time configurable, such that it is only in effect when required.

This builds on our insight that it is neither practical nor required to provide cache side-channel resilience for all the code in the workload. This additional security guarantee is only required for security-critical execution, which is a minority of the workload (Assumption A2), and usually isolated in a Trusted Execution Environment (TEE) (Assumption A1). Thus, we require to provide a cache architecture that provides non-isolated execution with conventional cache utilization (with no performance costs), and simultaneously side-channel-resilient cache utilization (with a tolerable performance degradation) only for the smaller portion of the execution workload that is security-sensitive and isolated. We also require that our architecture is portable, can be easily deployed, and is backward compatible when a system does not support it. We summarize these requirements below:

R1 Strong side-channel resilience guarantees between the isolated and non-isolated execution domains, sufficient to

thwart typical contention-based and access-based cache attacks

R2 Dynamic and scalable cache isolation between multiple different isolation domains

R3 Addressing the cache performance/security trade-off by configuring the non-isolated/isolated workload balance (compliant with how TEEs are intended and designed to be used) such that the performance of the non-isolated execution workload is not degraded

R4 Usability: backward-compatible, architecture-agnostic, no usage restrictions and no code modifications required

Next, we present the high-level construction of HYBCACHE in Section 4.2 and its microarchitecture in more detail in Sections 4.3 and 4.4.

4.2 High-Level Idea

In HYBCACHE, a subset of the cache, named *subcache*, is reserved to form an orthogonal isolated cache structure. Specifically, $n_{isolated}$ *cache ways* within the conventional *cache sets* form the *subcache*. While these *subcache* ways are available for the NI-Domain to utilize, the I-Domains are restricted to utilize *only* these *subcache* ways. However, the I-Domains utilize this *subcache* in a fully-associative way and using a random-replacement policy. In doing so, all mutually distrusting processes executing in the I-Domains can share the *subcache* without leaking information on the actual memory locations they access. Since these *subcache* ways are not reserved exclusively for isolated execution and can also be utilized by non-isolated execution with least priority, the NI-Domain still retains unaltered cache capacity usage and non-degraded performance.

The key purpose of HYBCACHE, unlike existing defenses, is to selectively enable side-channel-resilient cache utilization only for the I-Domains. Hence, only the isolated execution is subjected to the resulting performance overhead, while still maintaining conventional cache behavior and performance for the NI-Domain, as outlined in Requirement R3. We describe next the architecture of HYBCACHE and how it achieves this.

4.3 Controller Algorithm

HYBCACHE modifies how memory lines are mapped to cache entries for the I-Domains. $n_{isolated}$ ways (at least a way in each set) of the conventional set-associative cache are designated to the orthogonal *subcache*. Cache lines are mapped fully-associatively to the *subcache* entries and evicted and replaced in the *subcache* using a random replacement policy. This means that a given memory line can be cached in any of the $n_{isolated}$ entries. This breaks the deterministic link between memory addresses and their corresponding cache locations, thus defeating an attacker that attempts to infer the victim's memory accesses by triggering and observing contention in a particular cache set.

Figure 1 illustrates how the HYBCACHE controller manages cache requests. HYBCACHE supports multi-core processors with simultaneous multithreading (SMT) and assumes that each process is assigned an `IsolationDomainID` (`IDID`) that identifies whether the process is in an I-Domain (and which isolation domain) or in the NI-Domain. Any incoming cache request is accompanied by the `IDID` of the issuing process. In Ⓐ, HYBCACHE controller queries the `IDID` of the cache request and the request is serviced accordingly. If it is in the NI-Domain, the complete cache is queried conventionally using the set index and tag bits of the requested address to locate the cache set and line respectively (Ⓑ & Ⓒ). If a match is found, the controller checks whether the cache line was found in one of the *subcache* ways in Ⓓ. Recall that these ways are not reserved exclusively for isolated execution, i.e., they can be used by non-isolated execution but with least priority in case a cache set becomes over-utilized. Therefore, if a matching cache line is found in one of these ways, the controller checks whether it was cached by an isolated or non-isolated process (Ⓔ). The requesting process can only hit and access the cache line if that line was placed by a process in the NI-Domain. Otherwise, it is not allowed to hit on it.

Checks in the controller are implemented to occur in parallel, i.e., all cache hits are generated in the same number of clock cycles (as well as cache misses), to eliminate respective timing side channels. In case of a cache miss, the memory block is fetched from main memory and cached in Ⓕ. The eviction and replacement are performed according to the deployed policy. All ways are available for eviction, including the *subcache* ways to provide the NI-Domain execution with unaltered cache capacity. However, the usage of the *subcache* ways by the I-Domains is considered while recording the recency of accesses to the cache ways to make it least likely to evict a line from one of the *subcache* ways if it is recently used by an I-Domain process.

If the cache request is issued by an I-Domain process, it is serviced by querying only the *subcache* (Ⓖ). The *subcache* deploys fully-associative mapping, and is thus queried by a lookup of all the ways using the (cache line address bits - block offset bits) as tag bits (Ⓗ) and simultaneously querying that the line belongs to an I-Domain (since these ways may also be used by the NI-Domain) and that it was placed by a process with the same `IDID` (Ⓘ). Otherwise, a cache miss occurs. Disallowing I-Domain processes from hitting on cache lines originally placed by processes in other I-Domains provides dynamic isolation between an unlimited number of mutually distrusting processes that share memory. In case of a miss, any of the *subcache* ways is randomly selected and its cache line is evicted and replaced by the memory block fetched from main memory (Ⓙ). The random replacement policy considers all *subcache* ways equally, even those occupied by the NI-Domain cache lines.

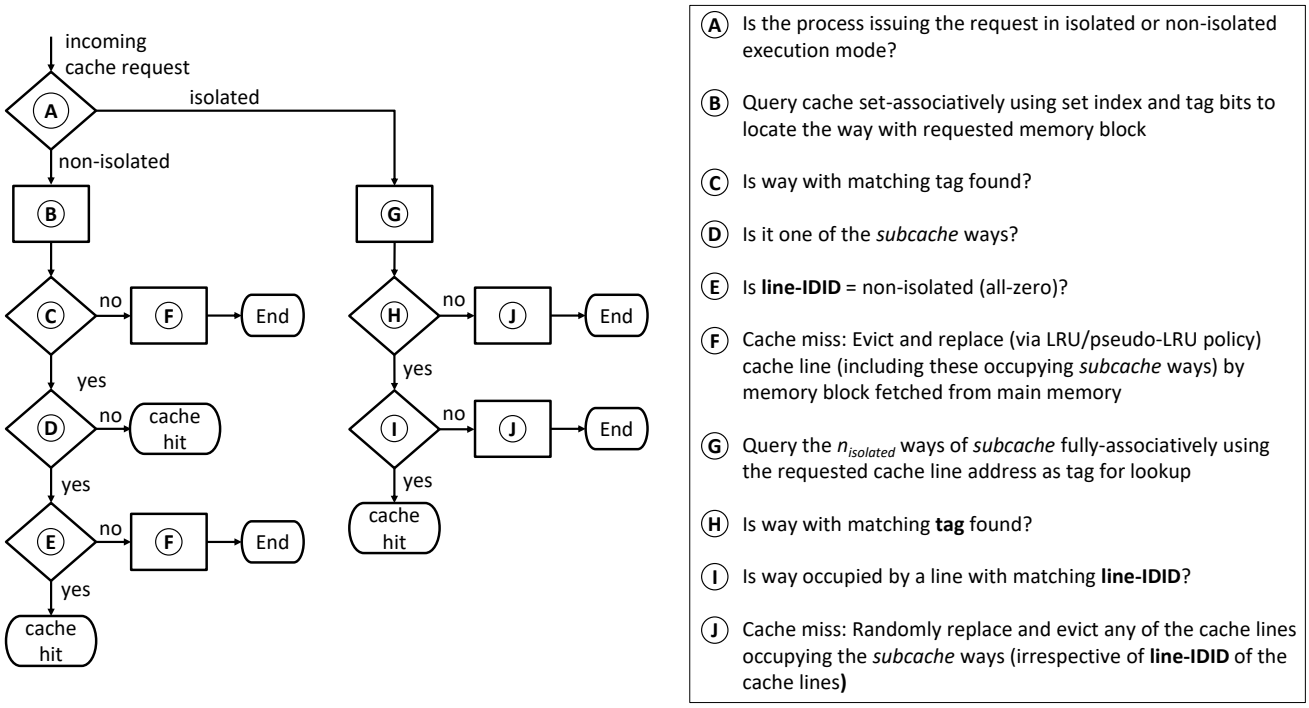


FIGURE 1: HYBCACHE controller policy

4.4 Hardware Microarchitecture

Figure 2 shows how HYBCACHE could be applied for a conventional cache hierarchy of a multi-core processor. The cache capacity available for the NI-Domain execution is unaltered, i.e., the conventional set-associative cache with all its sets and ways can be utilized by the NI-Domain.

At each cache level, way-based partitioning is used to reserve at least a way in each set (gray ways in Figure 2). These ways, combined, form the orthogonal *subcache* that the I-Domain execution is restricted to use. However, these *subcache* ways are *not* used exclusively by the I-Domain execution, i.e., the NI-Domain execution may use these ways in case a corresponding set is fully utilized and the least-recently-used (LRU) replacement algorithm requires to evict a cache line from a *subcache* way in this set. This ensures that the NI-Domain execution is provided with unaltered cache capacity and does not suffer performance degradation.

The *subcache* is fully-associative and deploys random replacement policy, i.e., a given memory block is always equally likely to be cached in any of the available ways. This breaks set-associativity and provides randomization-based dynamic isolation between different I-Domains while allowing flexible sharing of the *subcache* depending on the run-time utilization requirements of the isolated execution domains. Using the *subcache* fully-associatively further maximizes the utilization of its limited hardwired capacity.

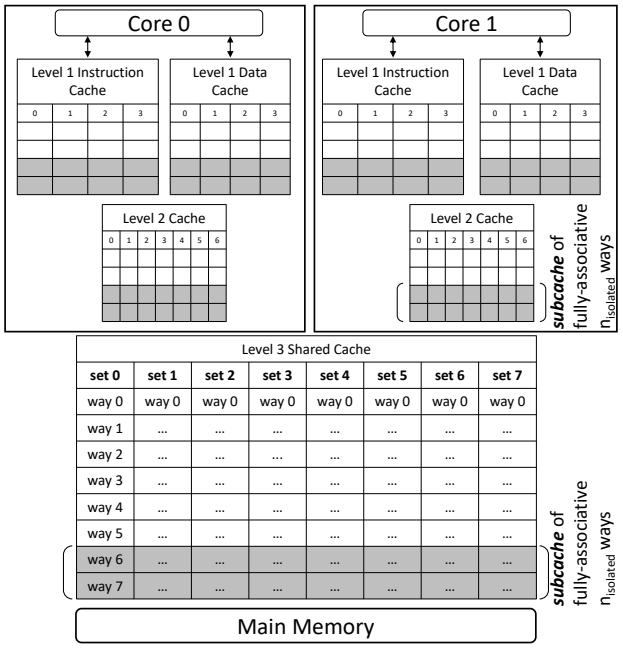


FIGURE 2: HYBCACHE hierarchy and organization

The $n_{isolated}$ ways that form the *subcache* are configured (hardwired) at design-time and cannot change at run-time, because these ways are members of both the primary cache as

well as the *subcache* as shown in Figure 3. It is not feasible to make $n_{isolated}$ run-time configurable, as this would require that *all* the ways are unreasonably wired in both a fully-associative and set-associative organization. Thus, only a small subset of $n_{isolated}$ ways (dark gray ways in Figure 3) is selected to form the *subcache*. Each of the *subcache* ways is augmented with `IsolationDomainID` (`IDID`) configuration bits to identify the isolation domain that placed an occupying cache line in the pertinent way. To provide any cache isolation at the microarchitectural level, a mechanism to bind owners/tags to cache lines is required, thus `IDIDs` are needed. We chose to configure 4 bits for the `IDID`, thus supporting 16 concurrent isolation domains, where an all-zero indicates the NI-Domain. The number of bits allocated in HYBCACHE for `IDID` is a hardware design decision. Increasing the number of designated bits would increase the number of maximum concurrent isolation domains that HYBCACHE can support. However, other metrics such as area overhead and power consumption come into play in this design trade-off.

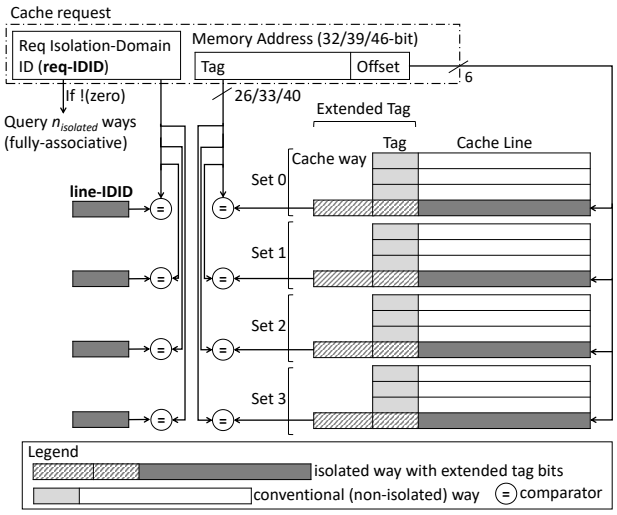


FIGURE 3: HYBCACHE hardware microarchitecture

The *subcache* ways are augmented with an extended tag bits storage (dashed dark gray tag bits of the dark gray ways in Figure 3). When queried fully-associatively (for the I-Domains), all bits, except the offset bits (6 bits for byte-addressable 64B cache line), of the requested address are compared with the extended tag bits of the *subcache* ways to locate a matching cache line. For the NI-Domain, the *subcache* ways are queried set-associatively with the rest of the cache (conventionally), where the request tag bits are compared only with the non-extended tag bits of the *subcache* ways within the located cache set.

4.5 Software Configuration

Abstraction and Transparency. The hardware modifications required for HYBCACHE are transparent to the software and abstracted from it. The trusted software (or hardware) component of the incorporating platform is only required to interface with the HYBCACHE controller to communicate the isolation domain of each incoming cache request. However, HYBCACHE does not stipulate or restrict how these isolation domains are defined and communicated, thus leaving it to the discretion of the system designer to identify how HYBCACHE can be integrated with the comprising architecture.

Isolated Execution. HYBCACHE enables the dynamic isolation of the cache utilization of different isolation domains by using the `IDID` of the process that issues the cache request being serviced. The means by which the isolation domains are defined, generated, and communicated is dependent on how the trusted execution and isolation is deployed. We design HYBCACHE such that it is seamlessly compliant with any trusted execution environment (TEE) where isolation domains (across different processes, cores, containers, or virtual machines (VMs)) are either software-defined by a trusted OS (thus requiring kernel support) or hardware/firmware-defined in case the OS is not trusted (such as in SGX). Different isolation domains can be defined across different isolated address space ranges such as in SGX enclaves, across processes such as in TrustZone normal/secure worlds or by standard inter-process isolation, or even across different groups of processes or different virtual machines.

HYBCACHE is agnostic to the means of defining the `IDIDs` of different isolation domains, and complements any form of isolated execution environment in place to provide it with cache side-channel resilience. If the kernel is trusted, kernel support is required to assign an `IDID` (or an all-zero `IDID` for a non-isolated process) to each process according to its isolation domain. The `IDID` bits can be added as an additional process attribute in each process's process control block (PCB). Otherwise, the trusted hardware or firmware would assign the isolation domains. HYBCACHE assumes that some mechanism of isolation is already enforced for security-critical code that it can leverage to provide the cache-level isolation. We argue why this is reasonable in Assumption A1. Nevertheless, if this is not the case, then isolation domains need to be explicitly defined by the developer if s/he wishes to protect particular code against cache-based side-channel attacks. While HYBCACHE is focused on protecting user code, in principle, kernel code can also be protected by allocating it to an isolation domain.

Backward Compatibility. Similar to processor supplementary capabilities such as Page Attribute Tables (PATs) and Memory Type Range Register (MTRR) for x86, HYBCACHE supports providing side-channel-resilience on-demand while

retaining backward compatibility. HYBCACHE only effectively provides side-channel resilience for the cache utilization of execution when processes are assigned different IDIDs that are communicated with each cache request. Otherwise, from a software perspective, HYBCACHE is identical to a conventional cache architecture. If no isolation domains are assigned to the different processes by the trusted kernel or trusted hardware, HYBCACHE is designed to assign an all-zero IDID by default to incoming cache requests and all execution is treated as non-isolated (see Figure 1) with cache-based side-channel resilience disabled. Only when kernel support is provided (or trusted hardware or firmware in case of SGX) does HYBCACHE behave differently for different isolation domains and provides its side-channel resilience capability.

Shared Memory Support. HYBCACHE supports, by design, that different isolation domains can share read-only memory, usually in the form of shared code libraries, without sharing the corresponding cache lines. This results in having multiple copies of the shared memory kept in cache (multiple cache entries), enforcing that cache entries are not shared between mutually distrusting code. Data coherence is also not a problem, in this case, since this is read-only memory. We elaborate in Section 5 how this effectively mitigates access-based side-channel attacks.

Conventional access to shared writable memory, on the other hand, between different isolation domains is disallowed by design in HYBCACHE, as this makes the victim process vulnerable to access-based attacks and would undermine cache coherence. In order to provide input and output functionality to isolated code, HYBCACHE provides special *I/O move instructions*. These allow code in an I-Domain to transfer data between a CPU register and a memory region (assigned an all-zero IDID when cached) that is designated exclusively for shared memory between processes belonging to different I-Domains. These special instructions are meant to be used to transfer data between domains only through this designated memory. In practice, we expect them to be used only in frameworks like the SGX SDK or a trusted kernel. If code in an I-Domain incorrectly accesses this memory region using regular instructions, or accesses its own memory using these special instructions, this could be disallowed, i.e., detected and blocked by the hardware or microcode, e.g., the MMU. This prevents inserting duplicated writable cache entries which can disrupt cache coherency, while ensuring that HYBCACHE's security guarantees still apply to any access performed using regular instructions.

5 Security Analysis

In the following, we evaluate the effectiveness of HYBCACHE with respect to the security requirements we outlined in Section 4.1. We show that HYBCACHE achieves these security guarantees by mitigating the following leakages:

S1 Malicious software running in an I-Domain or NI-Domain cannot flush or perform a cache hit on a cache line belonging to a different I-Domain.

S2 Malicious software running in an I-Domain or NI-Domain cannot pre-compute and construct an eviction set that selectively evicts a non-trivial subset of the cache lines belonging to a different I-Domain. Moreover, the set of the attacker's cache lines which can be evicted by the victim's lines does not depend on the addresses accessed by the victim.

S3 Cache hits generated by software in an I-Domain cannot be observed by software running in a different I-Domain or NI-Domain. Cache misses generated by software in an I-Domain can still be indirectly observed by malicious software running in a different I-Domain or NI-Domain, but the malicious software learns no information (e.g., memory address) about the access besides whether a cache miss has occurred.

5.1 S1: Absence of Direct Access to Cache Lines

Access-based attacks, like Flush + Reload [29, 78], Flush + Flush [26], Invalidate + Transfer [35], Flush + Prefetch [25], and Evict + Reload [27], require the attacker to have direct access to the victim's cache lines, normally as a result of shared memory between processes (e.g., shared libraries). As an example, Flush + Reload works by flushing shared cache lines and monitoring which lines the victim accesses and brings back into the cache. HYBCACHE mitigates this class of attacks by preventing shared cache lines between the attacker and victim, as we explain in the following.

Shared Read-Only Memory. Read-only memory is shared between different processes in case of shared code libraries. HYBCACHE provides support for shared read-only memory (Section 4.5), while fundamentally disallowing that any cache line is shared across different I-Domains. Execution within one domain can only access cache lines brought into the cache by the same domain. Separate (potentially duplicate) cache lines are maintained for each domain; flushing and reloading cache lines only impacts those owned by the attacker's domain and cannot influence any other I-Domain or leak any information on its cache lines. Having duplicate cache lines for read-only memory pages does not disturb cache coherency because it is read-only.

Shared Writable Memory. Shared writable memory between mutually distrusting domains is disallowed by design with HYBCACHE. Code in an I-Domain can only exchange data with another isolation domain through the special I/O

move instructions, which transfer data between the CPU registers and memory in the NI-Domain that is designated for shared communication (see Section 4.5). Incorrect usage of those instructions or incorrect access to this designated memory region could be detected and blocked by the MMU to prevent potential cache coherency disruption due to duplicate writable cache entries. However, HYBCACHE still enforces that every cache line only belongs to one domain. Since cache lines always belong to one specific I-Domain or the NI-Domain, code in a domain cannot flush or perform a cache hit on a different domain's cache lines (S1), and attacks that rely on those capabilities are thus impossible.

5.2 S2: Impossibility of Pre-Computed Eviction Set Construction

Without direct access to the victim's cache lines, attackers resort to contention-based attacks, like Prime + Probe [34, 38, 54, 61, 77], Prime + Abort [15], and Evict + Time [23, 61]. In these attacks, the attacker pre-computes and constructs an eviction set which ensures eviction of a specific subset of the victim's cache lines, e.g., lines that belong to a specific set in a set-associative cache. The attacker process first accesses the whole eviction set, thus ensuring the victim's cache lines are evicted. After a waiting interval, it then checks if its whole eviction set is still in cache by timing its own memory accesses to this set, thus detecting if the victim accessed any of the cache lines of interest. For a conventional set-associative cache, this is possible because of a fixed set-indexing, which can be directly determined from the target address of interest.

HYBCACHE protects I-Domains from such attacks by disabling the set-associativity of the reserved *subcache* entries when they are used by isolated execution: when a memory address is accessed by the isolated victim process, the cache line will be stored in any entry chosen randomly from the whole *subcache* and not from a specific set. The random replacement policy for isolated execution ensures that any of the *subcache* entries is chosen using a discrete uniform distribution, i.e., with an equal and independent probability every time, so the attacker has no means of identifying deterministically and reproducibly which cache set (or entry) will be used to cache a particular memory access of the victim. In order to ensure that a specific cache line of the victim is evicted, the attacker can only evict all lines in the *subcache*, but s/he cannot selectively evict a non-trivial subset of the victim's cache lines. Moreover, the set of the attacker's cache lines which can be evicted by the victim's lines does not depend on the addresses accessed by the victim (S2). As a consequence, attacks that rely on these capabilities are no longer possible. This holds whether the attacker process is running in an I-Domain or NI-Domain, as long as the victim process is in an I-Domain (Requirements R1 and R2).

5.3 S3: Observable Cache Events

Software running in an I-Domain can only hit on cache lines belonging to the same I-Domain. These cache hits generate no changes to the cache state, thus, they are unobservable by an attacker in a different I-Domain or in the NI-Domain.

Cache misses generated by software in an I-Domain evict a random cache line, which may belong to a different I-Domain or the NI-Domain. Malicious attacker code can then periodically observe how many of its lines are evicted and infer the number of cache misses the victim process is experiencing. The attacker can further use this information to infer the size of the victim's working set, i.e., the number of cache lines in the *subcache* currently belonging to the victim.

This cache occupancy channel is the only side-channel leakage that is not mitigated by the HYBCACHE construction, which is inherently available in any cache architecture where the attacker and the victim processes compete for entries in shared cache resources. It can only be effectively blocked by strict cache partitioning, which we deliberately do not provide in the HYBCACHE construction. This allows different isolation domains to still compete for cache entries, thus preserving maximum and dynamic cache utilization and unaffected performance for non-isolated execution, as our performance evaluation shows in Section 6.1. Note that, due to S2, the information inferred by the attacker from observing this remaining leakage, is effectively reduced to only knowing the working set size at any point in time.

Leveraging this side channel to infer further information and mount an attack in typical settings is not trivial. The victim may evict its own lines when it experiences cache misses due to the random replacement policy. This would not effect a difference in the cache state for the attacker, which complicates the attacker's bookkeeping. Moreover, observations are severely hindered when any other software is concurrently running besides the attacker and the victim processes. Finally, standard software hardening techniques can be applied to mitigate attacks to code implementations that are particularly sensitive to this attack. Furthermore, exploiting this side channel to leak data has not been shown in practice. A recent attack [67] leverages the cache occupancy side channel to infer which website is open in a different browser tab (under the strong assumption that no other tabs are open); however, it does not leak any user data. Cache activity masking is suggested as one of the countermeasures to the attack. Implementing cache activity masking for HYBCACHE is feasible and independent of our cache architecture.

Since the attacker aims to maximize its information and cannot observe cache hits, s/he can attempt to evict all *subcache* entries in order to maximize the number of misses experienced by the victim. As we discuss later, evicting the whole *subcache* takes time for an attacker in either the NI-Domain or in a I-Domain. An unprivileged attacker is unable to pause the victim's execution; thus, the attacker can only measure the

cache usage with limited granularity. However, a privileged adversary, like a malicious OS in the case of an SGX enclave, can stop and restart the victim arbitrarily and leverage tools like SGX-Step [12] to observe the victim's cache usage with fine granularity. HybCache does not mitigate such an attack by construction. However, mitigating it is only possible by strict cache partitioning and the resulting performance costs. We emphasize that we make an intentional design decision in HybCache to allow isolation domains to dynamically compete for cache entries for maximum cache utilization and unaffected performance for non-isolated execution. A HybCache construction that dynamically allocates a dedicated *subcache* for each isolation domain would block this leakage and mitigate attacks that rely on it.

Non-isolated Attacker Process. If the attacker process is in the NI-Domain, in order to guarantee eviction of the whole *subcache* it must fill up all ways in every cache set, including the *subcache* ways. Therefore, the attacker process must construct an eviction set that is as large as the entire cache capacity. A typical data L1 cache holds 512 cache entries. In our experiments, probing (accessing and measuring access latencies) of 512 cache lines takes approximately 30 000 CPU cycles, i.e., a little over 8 µs.[2] For larger caches, such as the LLC, it is not even feasible to mount Prime+Probe attacks by probing the entire cache. The adversary is required to pinpoint a few cache sets that correspond to the relevant security-critical accesses made by the victim and monitor these only [54].

Isolated Attacker Process. If the adversary is in a different I-Domain than the victim process, it still cannot control cache eviction of particular target addresses specifically. Both attacker and victim processes are isolated and can only use the *subcache* ways. Thus, an adversary aiming to perform controlled eviction can only try to evict the entire *subcache*. Because the *subcache* is fully-associative with random replacement, evicting the entire *subcache* requires an eviction set much larger than the *subcache* capacity. We argue below that this is not easier than probing the entire L1 cache (in case the attacker is non-isolated), for instance, even though the *subcache* is significantly smaller. Moreover, it can be only guaranteed up to a certain level of probabilistic confidence. This can be represented statistically by the coupon collector's problem, where coupons are represented by entries in the *subcache*. Let $N_{accesses}$ be the total number of accesses needed to evict all the *subcache* entries n and n_i be the number of accesses needed to evict the i-th way after i-1 ways have been evicted. Both $N_{accesses}$ and n_i are discrete random variables. The probability of evicting a new way becomes $\frac{(n-(i-1))}{n}$. The expected value and variance of $N_{accesses}$ are

$$\mathbb{E}(N_{accesses}) = n \cdot \mathrm{H}_n \qquad \mathbb{V}(N_{accesses}) \approx \frac{\pi^2}{6} \cdot n^2$$

H_n denotes the n^{th} harmonic number. For $n = 128$ *subcache* entries, an average of 695 memory accesses (each mapping to a different 64B cache line) is needed to evict the *subcache* with a variance of $\approx 26\,951$. This is comparably more than the 512 accesses required to probe the entire typical L1 cache if the attacker process is not isolated (see above). Moreover, with such a large variance, significant variations in the number of $N_{accesses}$ required are expected from the mean $\mathbb{E}(N_{accesses})$ every time this eviction process is repeated.

6 Evaluation

Cache	Size	Associativity	Sets
L1	64 KB	8-way associative	128
L2	256 KB	8-way associative	512
L3	4 MB	16-way associative	4096

TABLE 1: Cache hierarchy used in our evaluation

Mix	Components
`pov+mcf`	povray, mcf
`lib+sje`	libquantum, sjeng
`gob+mcf`	gobmk, mcf
`ast+pov`	astar, povray
`h26+gob`	h264ref, gobmk
`bzi+sje`	bzip2, sjeng
`h26+per`	h264ref, perlbench
`cal+gob`	calculix, gobmk
`pov+mcf+h26+gob`	povray, mcf, h264ref, gobmk
`lib+sje+gob+mcf`	libquantum, sjeng, gobmk, mcf

TABLE 2: Benchmark mixes used in our evaluation

HybCache is architecture-agnostic and applicable to x86, ARM or RISC-V. We performed our performance evaluation of HybCache on a gem5-based [9] x86 emulator. We evaluated the hardware overhead for an RTL implementation that we implemented to extend an open-source RISC-V processor Ariane [62]. For our prototyping, we applied HybCache to L1, L2, and LLC. We describe our evaluation results next.

6.1 Performance Evaluation

To evaluate HybCache, we chose eight mixes of programs from the SPEC CPU2006 benchmark suite, which are used in the literature[3] [36, 76], shown in the upper part of Table 2.

[2] We ran this experiment on an Intel i7-4790 CPU clocked at 3.60 GHz.

[3] [76] also uses a ninth mix, `dea+pov`, which fails to run on gem5.

Two-Process Mixes. In order to evaluate the impact of isolating one process in the context of an SMT processor, we configure gem5 to simulate two processors connected to a single three-level cache hierarchy, whose parameters are shown in Table 1. The caches have the latencies used in [76].

For each mix, we first isolate one process, then the other, and we compare the performance of those processes to a third run in which neither process is isolated. We make either 2 or 3 of ways per set usable by the isolated execution processes. The replacement policy for non-isolated processes is LRU. Like in [76], we let gem5 simulate the first 10 billion instructions of each process in order to let the process initialize, then we measure the performance of one additional billion instructions. We measure the performance overhead as the relative change in the instructions-per-cycle (IPC), i.e., the ratio between instructions executed and CPU cycles required. A *positive* overhead represents a *decrease* in performance.

Figure 4 reports the IPC overhead of each program when running in isolation mode, while the other member of the mix runs in normal mode, for 2 or 3 isolated ways. The geometric mean of the positive overheads is 4.95% with 2 isolated ways and 3.47% with 3 isolated ways, with maximum overheads of 16% and 14% respectively for the `cal+gob` mix. For this mix, the overhead is due to a significantly increased L3 cache miss rate: the data miss rate jumps from 0.6% to 17.6%, while the instruction miss rate increases from 2.1% to 9.0%. The working set of `calculix` normally fits in L3 [36] but it does not in the *subcache*, hence the higher overhead. Since HYBCACHE is meant to protect only sensitive applications, which can be expected to be short-lived and only constitute a minority of the workload of a system, we consider those overheads easily tolerable. Figure 5 reports the IPC overhead for the member of the mix that is not isolated. In all cases the IPC overhead is not positive, i.e., the IPC is equal or better than the baseline, thus showing that HYBCACHE does not degrade the performance of non-isolated processes.

Four-Process Mixes. To demonstrate scalability, we also ran four-process mixes, shown in the bottom part of Table 2. We configured gem5 with four cores; two cores share an L1 and L2 cache, the other two cores share one additional L1 and L2, while L3 is shared by all cores. Isolated execution can use two ways per set. We isolated each member of the two mixes (the first eight bars in Figure 6), while the other three processes were running normally. Each isolated process has an overhead similar to that reported in the two-process mix experiments in Figure 4. Moreover, we also isolated two processes in each mix (last two columns in Figure 6). In this case, we measured increased overheads by up to 2 additional percentage points due to the additional competition for the *subcache*. However, those overheads are still easily tolerable given the security benefits and that they are only incurred by the isolated execution.

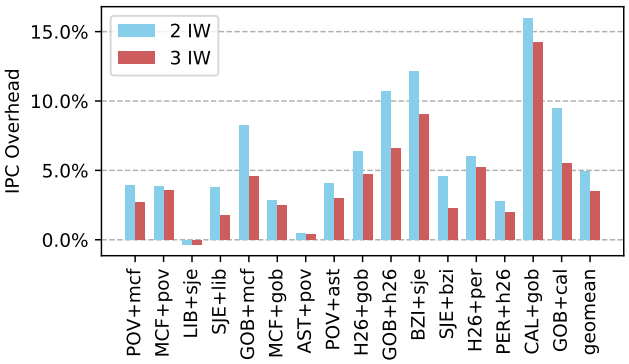


FIGURE 4: IPC overhead of each isolated process when 2 or 3 ways are available to isolated execution. Each pair of bars refers to a specific 2-process mix: the uppercase benchmark is isolated and the other is not.

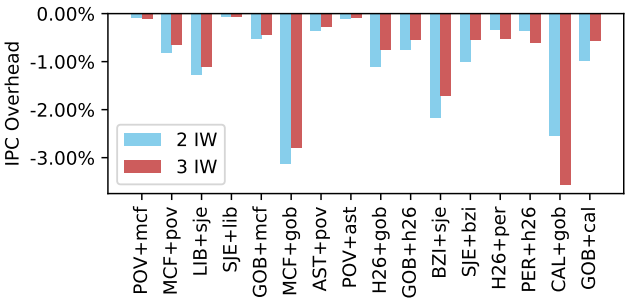


FIGURE 5: IPC overhead of each process when the other member of the mix is isolated. Each pair of bars refers to a specific 2-process mix: the uppercase benchmark is isolated and the other is not.

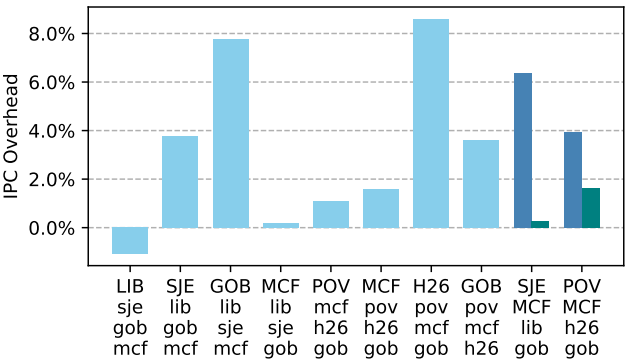


FIGURE 6: IPC overhead of isolated processes for 4-process mixes. The uppercase benchmarks are isolated and the others are not. The last two columns have two bars each since two process are isolated.

$n_{isolated}$	NAND2X1 Gates	Memory Overhead (Kb)
32	6114	0.34
64	12219	0.68
128	24563	1.3
256	48796	2.75
512	97830	5.5
1024	201792	11
2048	458300	22

TABLE 3: Logic and memory overhead estimates for fully-associative lookup of 46-bit addresses for different numbers of isolated cache ways (in any cache level).

6.2 Hardware and Memory Overhead

HYBCACHE requires additional hardware and memory for the fully-associative lookup of the *subcache* entries. We implemented the RTL for HYBCACHE and evaluated it for the hardware overhead for different number of isolated cache ways as shown in Table 3, irrespective of which cache levels this is applied to. While the overhead of the additional hardware is non-negligible, it is reasonable for a fully-associative cache lookup. Nevertheless, it diminishes in perspective with an 8-core Xeon Nehalem [1] of 2,300,000,000 transistors, for example. The logic overhead of HYBCACHE for 2048 fully-associative ways lookup is estimated at 1,833,200 transistors (NAND2X1 count $\times$ 4) which is 0.07% overhead to the Xeon Nehalem. For an 8-way 128-set cache, the memory overhead in our PoC for fully-associative mapping is 7 additional tag bits + 4 `IDID` bits per cache way. With respect to access latencies, the exact timing latency of lookups will eventually depend on the circuit routing but, in principle, for a parallel content-addressable memory lookup (as in our hardware PoC), accesses are performed in 2 clock cycles.

7 Discussion

Design and Implementation Aspects. HYBCACHE relies on a random-replacement cache policy combined with full-associativity to provide its dynamic isolation guarantees. The implementation of the random replacement policy is delegated to the hardware designer and considered an orthogonal problem. Cryptographically-secure pseudo-random number generators (CSPRNG) or even true hardware random number generators can be used and the seed can be changed as often as required. The output of the CSPRNG cannot be predicted if it is seeded with secret randomness at the start of every process. When the seed is changed, re-keying management tasks such as cache flushing and invalidation for the re-mapping are not required, unlike in recent architectures [63, 74]. This is because in HYBCACHE the randomness is only used for selection of the victim cache line, and not for locating existing cache lines in the *subcache*. Furthermore, we emphasize that CSPRNG design and implementations are an orthogonal problem to our work.

The "soft" cache partitioning of HYBCACHE is a generic concept and can be applied, in principle, to any set-associative structure. In this work, we apply it to the L1, L2, and L3 (LLC) caches, but it can also be applied selectively to only some of these cache levels or to the TLB as well, or to only some cache levels in only one or more cores in a multi-core architecture that become dedicated for allocating isolated execution. The choice of which cache structures to apply this to and how many ways to isolate in the *subcache* is delegated to the hardware designer, given that it is a more complex design decision with other metrics and trade-offs that come into play such as the size of the structure, power consumption, and logic overhead. The power consumption and timing overheads associated with building and routing a fully-associative cache lookup in VLSI are significant, but can be alleviated by leveraging emerging hybrid memory technologies such as DRAM-based caches [48] and STT-MRAM caches [30, 31]. In practice, applying HYBCACHE to the LLC or larger caches in general would be more expensive (in terms of hardware) than L1 and L2 caches, and strict partitioning might be applied instead for the LLC. Nevertheless, HYBCACHE can be, in principle, applied to sliced Intel LLCs. In each slice, a number of cache ways (*subcache*) is reserved for isolated execution. Any mapping from the `IDID` to the LLC slices can be used, such that lines from a particular `IDID` are allocated to a specific slice. Fully-associative lookups are thus only be performed on the *subcache* portion of a single slice, thus reducing the performance overheads and allowing scaling to high-core-count processors. The slice-mapping would be based only on the `IDID`, and thus it would not leak any information about the data address or value.

Other design decisions in HYBCACHE include the number of bits designated for `IDID` and thus the maximum number of concurrent isolation domains supported (see Section 4.4). To support more isolation domains (not concurrently) than the hardwired maximum, the cache lines of one domain can be flushed by the kernel or microcode at context switching while the next domain is switched in and is re-assigned the available `IDID`. Nevertheless, supporting too many isolation domains will result in increased cache utilization, and the overall performance will suffer. This is in line with conventional cache behavior, but is aggravated in HYBCACHE because isolated execution is only allowed to utilize the *subcache* portion. However, this violates our working assumption A2 that only the minority of the workload requires cache-level isolation.

We emphasize that cache-based side-channel leakage directly results from the design of the cache microarchitecture and, thus, it is reasonable to investigate the fundamental microarchitectural designs of caches for upcoming processor designs. While this does not address the problem for legacy systems, it provides an exploratory ground of ideas for upcoming processor designs. HYBCACHE is architecture-agnostic

and can be integrated with any processor architecture (we simulated it for x86 and implemented it for RISC-V). It is also compliant with any set-associative cache architecture independent of its hierarchy and organization, and whether it is virtually or physically indexed since no indexing is involved.

Intra-Process Isolation Support. HYBCACHE can also be extended, in principle, to provide *fine-grained* run-time configuration of the isolation domain *within* a process, e.g., between different threads within the same process. Besides kernel support, this requires an instruction extension to enable isolation of particular code regions or threads to different `IDIDs` or disable isolation altogether at run-time (reset its run-time `IDID` to all-zero). However, this requires the developer to identify and annotate security-sensitive code regions. Nevertheless, this is useful in practice since a process might not require cache-based side-channel resilience for its entirety but only for sensitive code such as cryptographic computations. This is a more generalizable approach that is easier and more directly applicable than implementing leakage-resilient variants for security/privacy-sensitive computations.

Deployment Assumptions. HYBCACHE assumes any TEE or trusted computing environment that is leveraged in compliance with their original design intent, i.e., that the much larger portion of the execution workload is not security-critical and only a smaller portion is security-critical and isolated in an I-Domain (A2). Otherwise, if the workload is equally balanced, the isolated execution subset would be restricted to a smaller partition of the cache and would incur a more than tolerable performance degradation especially if it is cache-sensitive. For HYBCACHE to be optimally advantageous, the workload distribution and allocation must be performed by the administrator such that the right balance of overall security and performance is achieved, as shown by the performance results in Section 6.1.

8 Related Work

We describe next the state of the art in existing defenses and their shortcomings that HYBCACHE overcomes.

8.1 Partitioning

Cache partitioning allocates to each process or security domain a separate partition of the cache, hence guaranteeing strict non-interference. Both software-based [20, 40, 51, 82] and hardware-based [24, 41, 72, 73] partitioning schemes have been proposed in recent years, where partitioning is either process-based or region-based.

Process-based partitioning. Godfrey [20] implements process-based cache partitioning using page coloring on Xen, which incurs a prohibitive performance overhead with increasing number of processes. SecDCP [72] is a way-partitioning scheme where each application is assigned a security class and cache partitioning between the security classes is dynamically managed according to the cache demand of non-secure applications. SecDCP is not scalable; selective cache flushing and repartitioning is required if the number of security classes exceeds that of allocated partitions and it may perform worse than static partitioning. Furthermore, both schemes do not support the use of shared libraries. CacheBar [82] periodically configures the maximum number of ways allocated to each process which unfairly impacts performance and cache utilization, and does not scale well with the number of security domains. DAWG [41] partitions the caches where different processes are assigned to different protection domains isolating cache hits and misses. The aforementioned schemes incur the performance overhead for the entire code, whereas HYBCACHE only enables side-channel resilience and the resulting performance overhead only for the isolated execution.

Sanctum [14] protects TEEs by flushing private caches whenever the processor switches between enclave mode and normal mode and partitioning of the LLC and assigning to each enclave a static number of sets. Sets allocated to an enclave can be used exclusively by the enclave and cannot be utilized by the OS. On the contrary, HYBCACHE allows for a flexible and dynamic sharing of cache resources between processes (thus improving performance), while preserving cache side-channel resilience for isolated execution.

Many cache partitioning and allocation schemes [37, 55, 64, 65, 75] have been proposed that focus on cache allocation mechanisms aiming to improve performance for multi-core caches. However, such schemes do not provide security guarantees. HYBCACHE addresses the security/performance trade-off by providing a configurable means to enable the side-channel resilience only for isolated execution while providing non-isolated execution with unaltered performance.

Region-based partitioning. These approaches split the cache into a secure partition reserved for security/privacy-critical memory pages and a non-secure partition for the remaining memory pages. STEALTHMEM [40] uses page coloring where several pages are colored and reserved for security-sensitive data and they remain locked in cache. CATalyst [51] leverages Intel's CAT (Cache Allocation Technology) [3] to divide the cache into secure and non-secure partitions and uses page coloring within the secure partition to isolate different processes' cache accesses to these pages. PLcache [73] locks cache lines and allocates them exclusively to particular processes such that the cache line can only be evicted by its process. However, overall performance and fairness of cache utilization are strongly impacted as the protected memory size increases in relevance to the total cache capacity. Moreover, with PLcache an attacker process may still infer the victim's memory accesses by observing that it is unable to access or evict cache lines (locked by a victim process) from a particular cache set.

Cloak [24] uses hardware transactional memory, such as Intel TSX [2], to protect sensitive computations by pre-loading the security-critical code and data into the cache at the begin-

ning of the transaction and any cache line evictions are detected by the transaction aborting. Cloak incurs prohibitively high performance overhead for memory-intense computations and requires the developer's strong involvement to identify and instrument security-sensitive code and split it into several transactions. Recent works have also explored the LLC inclusion property for defense schemes such as RIC [39] and SHARP [76]. However, both are architecture-specific, RIC requires coherence protocol modifications and cache flushing on thread migration, while SHARP requires modifications to the *clflush* instruction. HYBCACHE, however, is architecture-agnostic, and does not require cache flushing or modifications to coherence protocols or the *clflush* instruction.

8.2 Randomization

Introducing randomization involves introducing noise or deliberate slowdown to the system clock to hinder the accuracy of timing measurements as in FuzzyTime [32] and TimeWarp [57]. These techniques can only defeat attacks which rely on measuring access latency, but cannot prevent other attacks such as alias-driven attacks [28]. They compromise the precision of the clock for the remaining workload, thus affecting functionality requirements.

RPCache [73] randomizes the mapping of all memory lines of a protected application at a per-set granularity from their actual cache set to a randomly mapped cache set, by using a permutation table. NewCache [53] randomizes the mapping at a per-line granularity using a Random Mapping Table. Both RPCache and NewCache schemes do not scale well with the number of lines in the cache (not applicable for larger LLCs) and the number of protected domains. Random Fill Cache [52] mitigates only reuse-based cache collision attacks by replacing deterministic fetching with randomly filling the cache within a configurable neighborhood window whose size impacts the performance degradation incurred. It does not scale well with an increasing TEE size.

Time-Secure Cache [69] 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 re-keying is frequently required followed by cache flushing which requires complex management and impacts performance. CEASER [63] also uses a keyed indexing function but without the Process ID, thus also requiring frequent re-keying of its index derivation function and re-mapping to limit the time interval for an attack. A concurrent work, ScatterCache [74], uses keyed cryptographic indexing that depends on the security domain, where cache set indexing is different and pseudo-random for every domain but consistent for any given key. Thus, re-keying may still be required at time intervals to hinder the profiling and exploitation efforts of an adversary attempting to construct and use an eviction set to collide with the victim access of interest. HYBCACHE, on the other hand, leverages randomization by disabling set-associativity altogether and using random replacement for isolated execution. Every given memory address can be cached in any of the available *subcache* ways and placement is random and unpredictable; it varies randomly every time the same memory line is brought in cache.

9 Conclusion

In this paper, we proposed a generic mechanism for flexible and "soft" partitioning of set-associative memory structures and applied it to multi-core caches, which we call HYBCACHE. HYBCACHE effectively thwarts contention-based and access-based cache attacks by selectively applying side-channel-resilient cache behavior only for code in isolated execution domains (e.g., TEEs). Meanwhile, non-isolated execution continues to utilize unaltered and conventional cache behavior, capacity and performance. This addresses the persistent performance/security trade-off with caches by providing the additional side-channel resilience guarantee, and the resulting performance degradation, only for the security-critical execution subset of the workload (usually isolated in a TEE) by eliminating the fundamental causes of these attacks. We evaluated HYBCACHE with the SPEC CPU2006 benchmark and show a performance overhead of up to 5% for isolated execution and no overhead for the non-isolated execution.

Acknowledgments

We thank our anonymous reviewers for their valuable and constructive feedback. We also acknowledge the relevant work of Tassneem Helal during her bachelor's thesis. This work was supported by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the German Research Foundation (DFG) through CRC 1119 CROSSING P3, and the German Federal Ministry of Education and Research through CRISP.

References

[1] INTEL. Intel Xeon Processors. https://www.intel.com/content/www/us/en/products/processors/xeon.html, 2009.

[2] INTEL. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2016.

[3] INTEL. Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, 2016.

[4] Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/

01/reading-privileged-memory-with-side.html, 2018.

[5] 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.

[6] 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.

[7] 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.

[8] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Computer Architecture News*, 39(2), 2011.

[10] Joseph Bonneau and Ilya Mironov. Cache-collision Timing Attacks Against AES. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer-Verlag, 2006.

[11] 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.

[12] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-step. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution - SysTEX'17*. ACM Press, 2017.

[13] Victor Costan and Srinivas Devadas. Intel SGX Explained. Technical report, Cryptology ePrint Archive. Report 2016/086, 2016. https://eprint.iacr.org/2016/086.pdf.

[14] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, pages 857–874, 2016.

[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-free High-precision L3 Cache Attack Using Intel TSX. In *USENIX Security Symposium*, 2017.

[16] 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.

[17] 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.

[18] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[19] 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.

[20] Michael Godfrey. On The Prevention of Cache-Based Side-Channel Attacks in a Cloud Environment. Master's thesis, Queen's University, Ontario, Canada, 2013.

[21] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, 2017.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[28] 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.

[29] 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.

[30] Xiaochen Guo, Engin Ipek, and Tolga Soyata. Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2010.

[31] F. Hameed, A. A. Khan, and J. Castrillon. Performance and Energy-Efficient Design of STT-RAM Last-Level Cache. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2018.

[32] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[33] Intel. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[34] 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.

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

[36] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive Insertion Policies for Managing Shared Caches. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[43] Helge Klein. Modern multi-process browser architecture. https://helgeklein.com/blog/2019/01/modern-multi-process-browser-architecture/, 2019.

[44] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203*, 2018.

[45] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-channels. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2012.

[46] 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.

[47] 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.

[48] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. A Fully Associative, Tagless DRAM Cache. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2015.

[49] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[51] 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.

[52] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

[53] 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.

[54] 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.

[55] Wanli Liu and Donald Yeung. Using Aggressor Thread Information to Improve Shared Cache Management for CMPs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[56] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[57] 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.

[58] Matt Miller. Mitigating arbitrary native code execution in microsoft edge. https://blogs.windows.com/msedgedev/2017/02/23/mitigating-/, Jun 2018.

[59] 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.

[60] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. Technical report, arXiv:1703.06986 [cs.CR], 2017. https://arxiv.org/abs/1703.06986.

[61] 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.

[62] Pulp-Platform. Ariane RISC-V CPU. https://github.com/pulp-platform/ariane.

[63] Moinuddin K. Qureshi. Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[64] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006.

[65] Daniel Sanchez and Christos Kozyrakis. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[66] 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.

[67] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. *CoRR*, abs/1811.07153, 2018.

[68] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy managemen. In *USENIX Security Symposium*, 2017.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] Yuejian Xie and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

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

[77] 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.

[78] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security Symposium*, 2014.

[79] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. volume 7, pages 99–112. Springer, 2017.

[80] 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.

[81] 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.

[82] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. A Software Approach to Defeating Side Channels in Last-Level Caches. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

G

CHUNKED-CACHE: ON-DEMAND AND SCALABLE CACHE ISOLATION FOR SECURITY ARCHITECTURES

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

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

Ghada Dessouky, Alexander Gruler, Pouya Mahmoody, Ahmad-Reza Sadeghi, Emmanuel Stapf
Technische Universität 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 solution CHUNKED-CACHE addresses the security-performance trade-off practically and optimally by enabling efficient selective and on-demand utilization of side-channel resilience 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 minimum 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 [55], [1], [61], [57], [53], [66], [100], [32], [29], [28], [59], [4], [3], [88], [69], [90], [92], [16], [17], [83], [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 [39], [99], [36], [45], [35], [44], [47], [65], [33], [72], [37], [38], [98], [91]. In typical side-channel attacks [72], [44], [47], [65], [39], [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], [65], [72], [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 [35], [33], [14], [36], [37], 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 [42], [21] or ARM TrustZone [5], have been shown vulnerable to these attacks, thereby undermining their acclaimed privacy and isolation guarantees [14], [85], [70], [31], [60], [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 [27], [26], [56] to software- and hardware-based approaches that modify the cache organization itself. The latter can be broadly classified into either cache partitioning [30], [94], [52], [62], [23], [52] or randomization-based techniques [64], [89], [79], [80], [96], [87] that attempt to obfuscate the relationship between the memory address and the cache location it is mapped to.

More recently, various schemes for a randomized memory-to-LLC mapping, such as CEASER, ScatterCache, and Phantom-Cache [89], [79], [80], [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 [80] were able to undermine them. Consequently, enhancements to these defenses were proposed [80], only to be rendered ineffective again by yet another attack vector, e.g., weak low-latency cryptographic primitives [76], [10].

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], [58], [94], [52], [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 only a reasonable 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 [86] which are not mitigated by recent works [96], [23]), while maintaining the desired performance requirements.

To address this performance-security trade-off most optimally, 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 conventional way-based cache 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 run time 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 way-based partitioning 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 [55] and Meltdown [61], besides threatening a wide spectrum of platforms and architectures [60], [65], [44], [102], and even TEE architectures [14], [85], [70], [31], [60], [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 [39], [99], [36], [45], [35] where the target address is explicitly accessed and flushed.
- Conflict-based approaches [72], [44], [47], [99], [65], [98], [24], [33], [72], [38], [91], [7], [11] where the adversary triggers a controlled cache contention in the same cache set of the target address to evict the corresponding victim cache lines.

- Occupancy-based approaches [86] 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 [43], [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], [74] 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 [36].

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 [73], [75], [67], by injecting noise into the system [93], [41] or deliberately slowing down the system clock [40], [68]. However, workarounds have been found to create timers [84] or to perform attacks without relying on timers [25]. 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], [79], [80], [96], [87], [95], [64], [63] or 2.) cache partitioning to provide strict isolation [30], [51], [101], [62], [22], [34], [103], [48], [97], [58], [6], [94], [52], [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 have been shown to undermine such defenses [80], [12], [77], [76]. Moreover, many rely on cryptographic primitives which have been shown vulnerable to cryptoanalysis, while deploying more secure primitives would further degrade performance [10], [76].

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], [52], [34]. Several approaches do not directly cater for the use of shared libraries [30], [94], are architecture-specific [48], [97] or do not defend against occupancy-based attacks. Memory page coloring approaches [22], [51], [30] 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], [58], [13], [6] and industry solutions [42], [46], [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], [58]), 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 [42] or Sanctum [42], at the system bus in CURE [6] or by the Physical Memory Protection (PMP) unit in Keystone [58]. The access control mechanisms are either configured by microcode [42],

[46] 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], [58], [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 [46], 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], [58], [13], [6], [42], [46], [5].

With regard to cache side-channel attacks, we assume the adversary 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], [79], [80], [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 kept out of scope, since they are very specific to certain software implementations and have not been frequently shown.

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 [30], [94], [52], [62], [23], [95], [64], [89], [79], [80], [96], [87], we do not consider physical attacks on caches, e.g., physical side-channel attacks [54], fault injection attacks [9], and attacks that exploit hardware flaws [88], [49], [78]. 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).

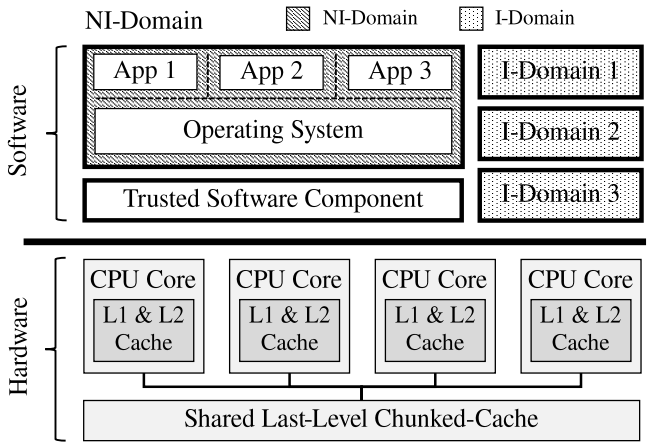


Fig. 1. Computing system with TEE architecture and CHUNKED-CACHE as the shared last-level cache.

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 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

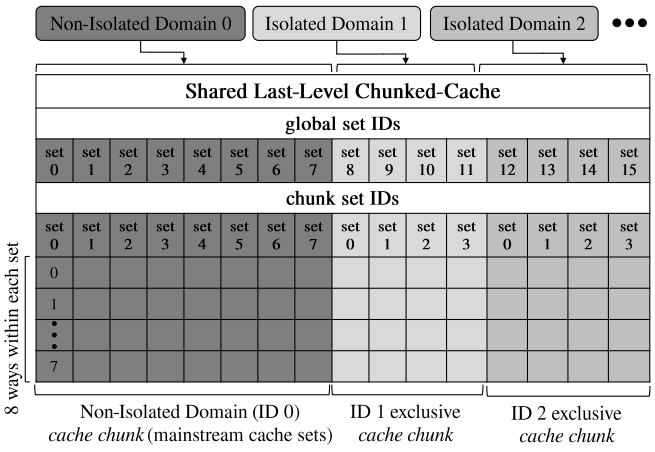


Fig. 2. CHUNKED-CACHE high-level design: each domain gets an exclusive *cache chunk* allocated on-demand.

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 guarantees 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 way-based partitioning schemes [62], [58], [6], [94], [52] 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 utilization, significantly improved performance, and enhanced scalability, as we show in Section VI.

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 [42], [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.

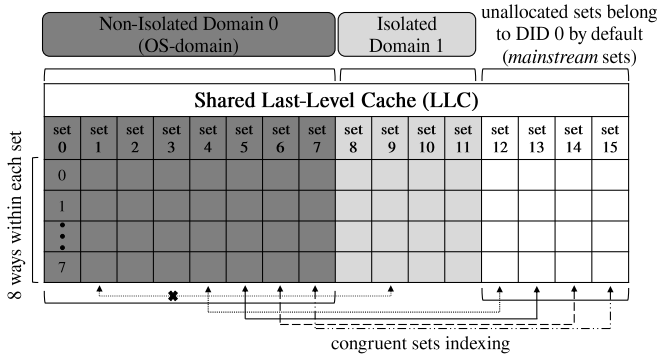
When an I-Domain is also sharing memory with the OS, the corresponding cache lines for the defined SHARED MEMORY regions are cached in the mainstream cache sets, and are to be accessed by both the owner domain and the OS. To support that, cache lines need to be tagged with an additional `SHARED` flag 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 that its cache line is mapped to. 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 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.



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.

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 a memory address is mapped to. 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.

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

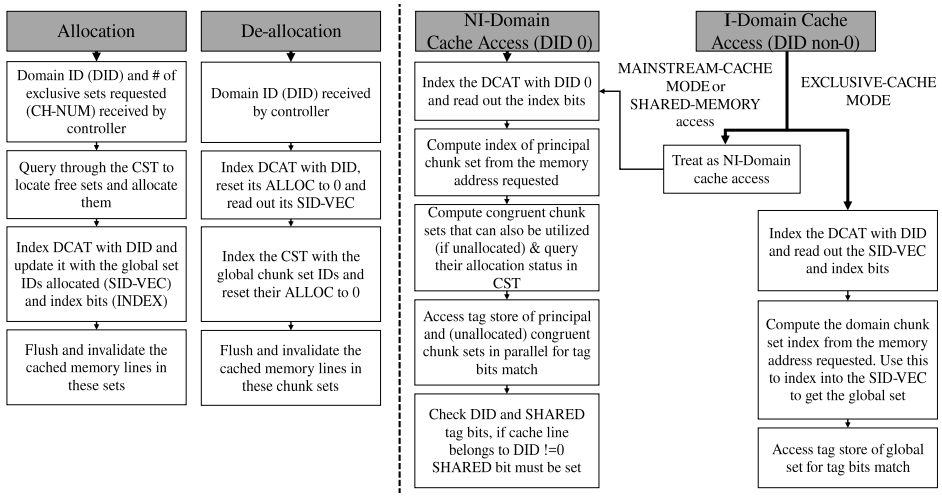


Fig. 4. CHUNKED-CACHE controller operations for cache chunk allocation, de-allocation and access control.

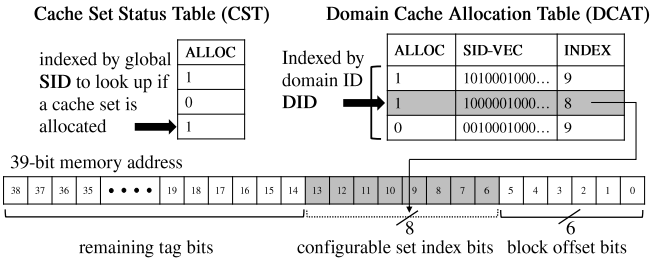


Fig. 5. CHUNKED-CACHE table structures.

allocated and that the maximum sets number per I-Domain is not exceeded. 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 in order 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.

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, DCAT is indexed with the all-zero ID to read out the index bits. The OS domain is assigned the least significant cache sets by default, thus the SID-VEC is 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 principal and congruent sets are looked up in parallel to locate a tag bit match (cache hit). In parallel, the DID and SHARED tag bits are also checked. 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 randomized cache line mappings [89], [79], [80], [96], [87], CHUNKED-CACHE provides well-grounded security guarantees which do not depend on weak cryptographic primitives. 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

136

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 [52]. 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.

CHUNKED-CACHE allows for a dynamic assignment of cache sets to I-Domains. Whenever the chunk set 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. 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 [79], [80], [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 webserver `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 [58], [6]. We define for our implementation that the maximum number of sets that can be allocated to any domain is

137

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 $\times$ 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 need to be 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 would incur a latency of 16,384 cycles and at the best case, CH-NUM cycles. Additionally, a fixed latency of 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.

138

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.

For de-allocation, we measure an overall latency of CH-NUM + 2 cycles, whereby 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 to invalidate or flush any other sets of the NI-Domain which would require re-caching them. This is one key design goal of CHUNKED-CACHE since it eliminates this performance overhead on 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 runtime which requires a context switch out of the I-Domain. The overhead CHUNKED-CACHE induces for the allocation/de-allocation remains negligible when compared with the general overheads of TEE architectures [22], [13], [58], [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 [79], [80], [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. Moreover, to also cover I/O-intensive workloads, we evaluate the impact of CHUNKED-CACHE on the widely used webserver `nginx`. We run our experiments for 1 trillion instructions before we

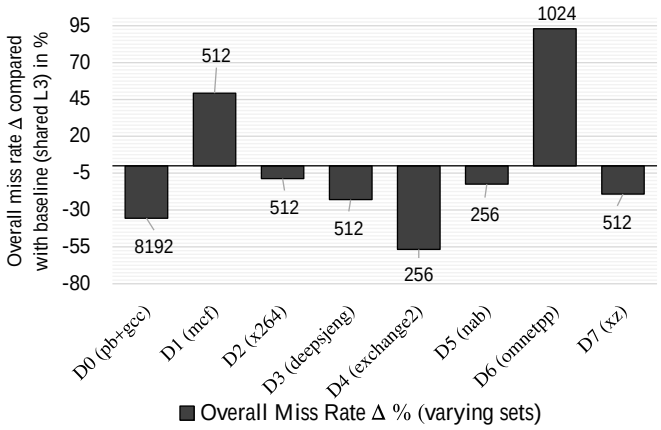


Fig. 6. Cache miss rate impact of CHUNKED-CACHE for SPEC benchmarks on a 8-domain setup; compared to a shared L3 cache.

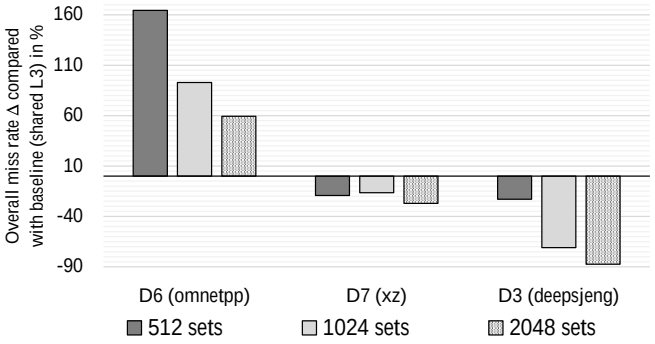


Fig. 7. Cache miss rate impact of CHUNKED-CACHE for SPEC CPU2017 benchmarks (varying sets); compared to a shared L3 cache.

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 [62] which uses Intel CAT [50], SecDCP [94], DAWG [52], Keystone [58] 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. All the following experiments were conducted on an x86 platform equipped with an Intel Xeon Silver 4215 CPU (2.50 GHz) and 186 GB RAM.

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 results 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 choose the number of sets by briefly analyzing the working set size of the benchmark running in each I-Domain, whereby bigger working sets get more cache sets assigned. 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.

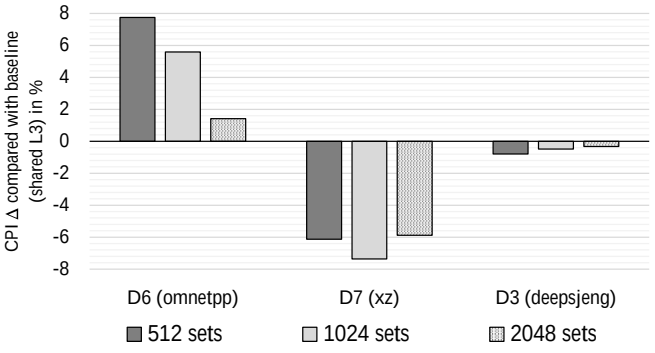


Fig. 8. CPI impact of CHUNKED-CACHE for SPEC CPU2017 benchmarks (increasing sets); compared to a shared L3 cache.

Additionally, to evaluate the impact of CHUNKED-CACHE on I/O-intensive workloads, we conduct experiments in which we run the `nginx` webserver 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 webserver 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.

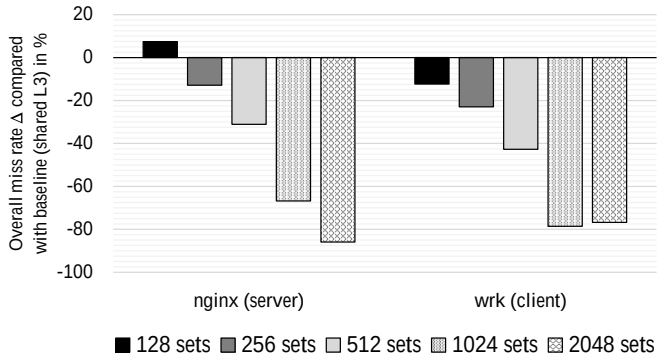


Fig. 9. Cache miss rate impact of CHUNKED-CACHE for `nginx` and `wrk` (increasing sets); compared to a shared L3 cache.

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 is preserved which was one of the key design goals of CHUNKED-CACHE.

To investigate this, we run experiments (same setup) in 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.

Comparison with Way-based Partitioning. We compare CHUNKED-CACHE to a way-based cache partitioning scheme which we implement on gem5, 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

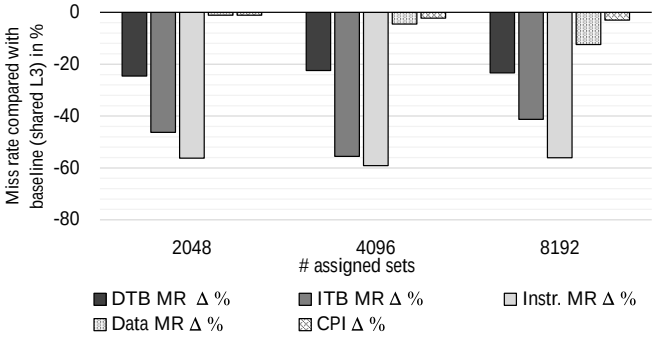


Fig. 10. Miss rate & CPI impact of CHUNKED-CACHE on the NI-Domain (increasing sets); compared to a shared L3 cache.

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.

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 [81] using Xilinx Vivado tools. While this processor does not provide an LLC, this is not necessary since we can still extend the

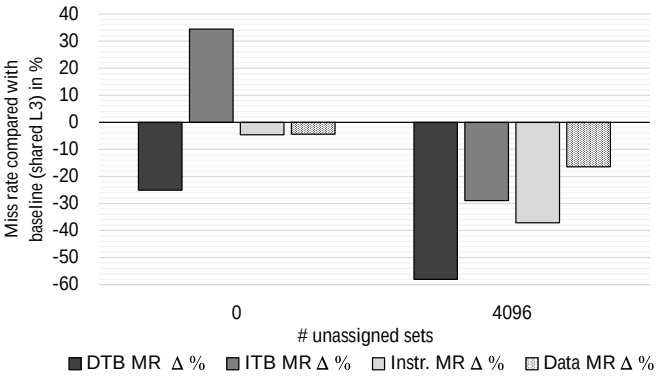


Fig. 11. Miss rate of CHUNKED-CACHE on the NI-Domain with varying number of unassigned sets; compared to a shared L3 cache.

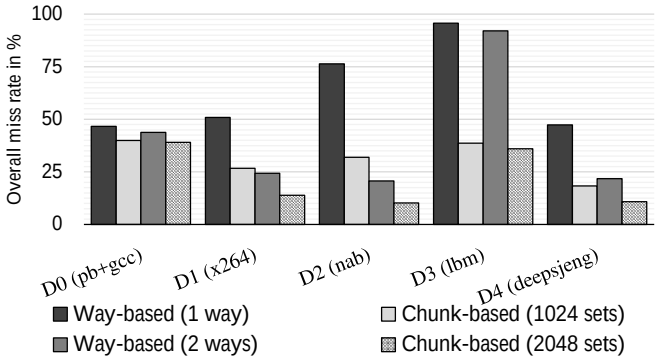


Fig. 12. Overall miss rate for SPEC CPU2017 benchmarks with CHUNKED-CACHE; compared to a way-partitioned cache.

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 in fact 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 $\approx$ 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 storage and area overheads are directly impacted by the various design/implementation trade-offs involved which can be configured differently in different implementations of CHUNKED-CACHE. For example, the CST storage required can be reduced if the number of sets always allocated to the NI-Domain is hard-wired, e.g. to 8,192 sets, such that these sets do not need to be tracked. Similarly, the DCAT storage overhead would also be significantly reduced since these sets would not be accounted for. The DCAT overhead can also be reduced significantly if contiguous set allocation is used instead, thus, reducing the length of SID-VEC, which is the largest overhead contributor. In our design and prototype, we decided against contiguous set allocation due to fragmentation that can eventually occur as the system continues to run.

In short, the capacity of these tables varies with 1.) the number of active parallel domains supported (overhead increases only linearly) 2.) the cache capacity they are tracking and its total number of sets, 3.) the maximum number of sets that can be allocated to a domain, and 4.) whether the sets for the NI-Domain are hard-wired to a certain number at design-time. The configuration of these trade-offs for a specific implementation directly impacts the incurred overheads.

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 thus diminish negligibly when ported to a significantly more complex multi-billion-transistor processor with a 3-level cache hierarchy which is the intended platform for CHUNKED-CACHE.

Power Consumption Overhead. CHUNKED-CACHE incurs a higher static leakage power consumption and dynamic consumption per access due to the additional tag bits per cache line (4-bit `DID` and 1-bit `SHARED` flag) as well as the CST and DCAT tables, though the LLC power consumption is largely dominated by the static leakage power. We estimate the power consumption overheads of CHUNKED-CACHE in 22nm technology using the CACTI-6.0 tool [71]. 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 [58] 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 [52] 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.

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.

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], [64], [63] 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 [79] 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 [80], 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 [80] 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], [77] with no solid future-proof 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 [76], [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 [82]. 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, at minimum hardware and memory costs.

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.

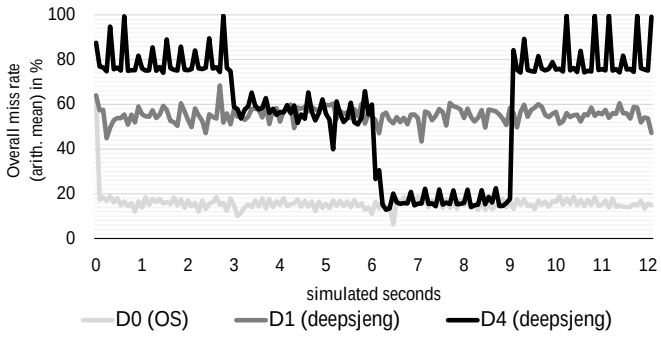


Fig. 13. Cache miss rate impact of CHUNKED-CACHE on the NI-Domain (D0) and I-Domains (D1, D4) when dynamically modifying the size of the cache chunk assigned to D1.

ACKNOWLEDGEMENT

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 Enclave. *arXiv preprint arXiv:2010.15866*, 2020.

[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.

143

[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*. USENIX Association, 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 *USENIX Security Symposium*, 2017.

[25] 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.

[26] 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.

[27] 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.

[28] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[29] 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.

[30] Michael Godfrey. On The Prevention of Cache-Based Side-Channel Attacks in a Cloud Environment. Master's thesis, Queen's University, Ontario, Canada, 2013.

[31] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on Systems Security*, 2017.

144

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[38] 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.

[39] 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.

[40] Wei-Ming Hu. Reducing Timing Channels with Fuzzy Time. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1991.

[41] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1(3-4):233–254, 1992.

[42] Intel. Intel Software Guard Extensions. Tutorial slides. https://software.intel.com/sites/default/files/332680-002.pdf. Reference Number: 332680-002, revision 1.1.

[43] Intel. Intel Integrated Performance Primitives Cryptography Developer Reference. 2019.

[44] 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.

[45] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2016.

[46] Kaplan et al. AMD memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[54] Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, 1996.

[55] 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.

[56] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic Quantification of Cache Side-channels. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2012.

[57] 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.

[58] 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.

[59] 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.

[60] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[61] 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.

[62] 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.

[63] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014.

[64] 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.

[65] 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.

[66] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[74] Mathias Payer. Hexpads: a platform to detect "stealth" attacks. In *International Symposium on Engineering Secure Software and Systems*, pages 138–154. Springer, 2016.

[75] Colin Percival. Cache missing for fun and profit, 2005.

[76] 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.

[77] Antoon Purnal and Ingrid Verbauwhede. Advanced profiling for probabilistic prime+ probe attacks and covert channels in scattercache. *arXiv preprint arXiv:1908.03383*, 2019.

[78] 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.

[79] Moinuddin K. Qureshi. Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

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

[81] Roa Logic BV. RV12. https://github.com/RoaLogic/RV12, 2020.

[82] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX Security Symposium*, 2021.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[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.

145

[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.

H

CURE: A SECURITY ARCHITECTURE WITH CUSTOMIZABLE AND RESILIENT ENCLAVES

[11] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclave. In USENIX Security. USENIX Association, 2021. Core Rank A*.

CURE: A Security Architecture with CUstomizable and Resilient Enclaves

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

Abstract

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

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

1 Introduction

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

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

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

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

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

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

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

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

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

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

2 System Assumptions

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

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

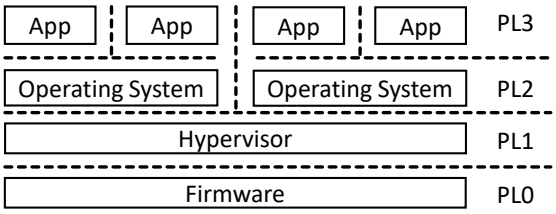


Figure 1: Software privilege levels (PL): user space, kernel space & dedicated levels for hypervisor & firmware.

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

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

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

3 Adversary Model

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

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

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

4 Requirements Analysis

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

4.1 Security Requirements (SR)

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

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

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

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

4.2 Functionality Requirements (FR)

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

at different privilege levels can be supported.

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

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

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

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

5 Design of the CURE Architecture

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

5.1 CURE Ecosystem

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

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

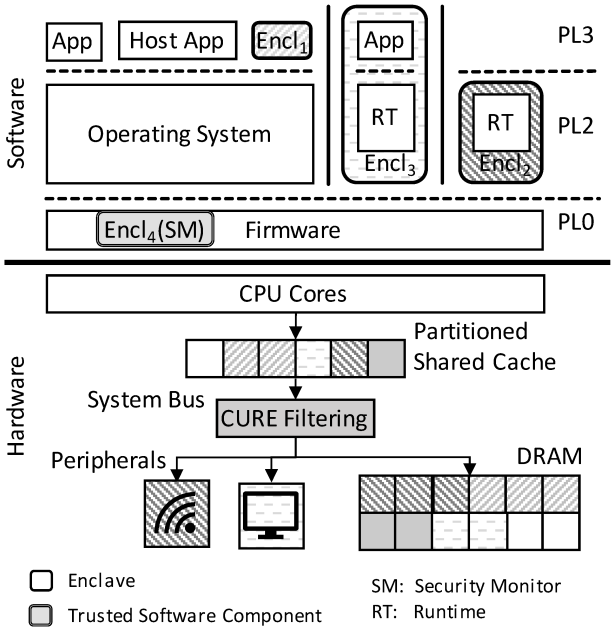


Figure 2: CURE privilege levels and enclave types, namely, user-space enclaves (Encl_1), kernel-space enclaves (Encl_2, Encl_3) and sub-space enclaves (Encl_4).

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

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

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

5.2 Customizable and Resilient Enclaves

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

5.2.1 Enclave Management

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

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

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

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

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

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

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

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

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

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

5.2.2 User-space Enclaves

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

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

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

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

5.2.3 Kernel-space Enclaves

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

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

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

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

5.2.4 Sub-space Enclaves

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

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

5.3 Hardware Security Primitives

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

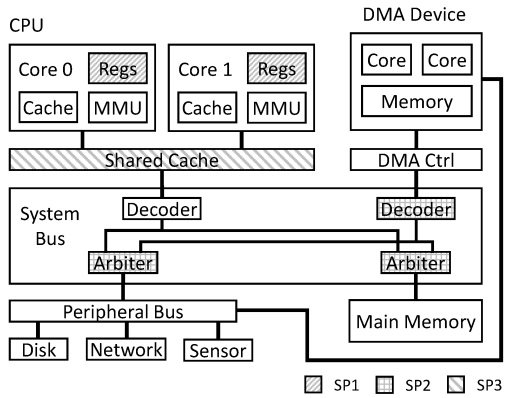


Figure 3: CURE Security Primitives (SPs), added at core register files (SP1), system bus (SP2) and shared cache (SP3).

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

5.3.1 Defining Enclave Execution Contexts (SP1)

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

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

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

5.3.2 Access Control on the Bus (SP2)

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

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

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

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

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

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

5.3.3 On-Demand Cache Partitioning (SP3)

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

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

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

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

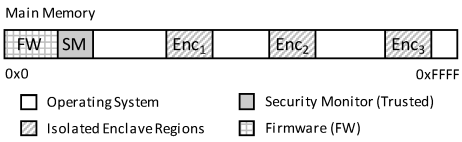


Figure 4: Physical memory layout of our CURE prototype.

6 Prototyping CURE on RISC-V

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

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

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

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

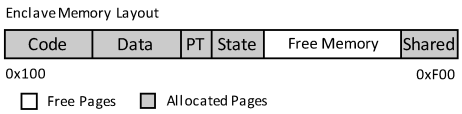


Figure 5: CURE enclave memory layout consisting of the code & data pages, page tables (PT), the enclave state (State) and the shared memory (Shared).

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

6.1 Software CURE Enclaves

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

6.1.1 Enclave Memory Layout

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

6.1.2 Security Monitor

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

6.1.3 User-space Enclaves

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

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

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

6.1.4 Kernel-space Enclaves

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

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

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

6.2 Hardware Security Primitives

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

6.2.1 Extending the TileLink Protocol

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

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

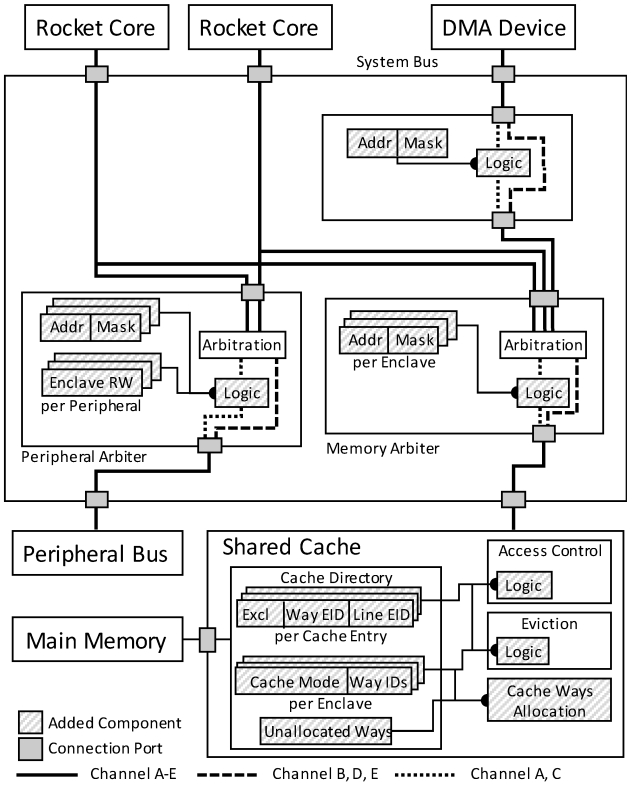


Figure 6: CURE prototype implementation using Rocket Chip.

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

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

6.2.2 System Bus Access Control

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

Performing access control. The added registers hold memory ranges defined by a 32-bit base address (`Addr`) and a 32-bit mask (`Mask`), and are used by the control logic to perform access control on every memory transaction using the `eid` and `address` signals. Access control is only performed on channels with a parent-to-child direction (channels `A` and `C`). At access violation, the transaction is redirected (with all-zero data) to an unused, zero-initialized memory region. Thus, all forbidden transactions write/read zeros to/from the unused memory region. An adversary enclave might fill L1 with malicious data which could get flushed with SM privileges during enclave context switch. To prevent this, we modify the core such that on every switch to the SM, the L1 is flushed before the `eid` register is set. We connect the system bus to the peripheral and interrupt bus. This allows the SM to configure the added registers and control logic, and trigger an interrupt upon access violation which is handled by the SM.
Memory arbiter. We add 15 registers to the memory arbiter, one for each enclave (13), the SM and the firmware. Each register defines the memory region assigned to each execution context. For the enclaves, the control logic verifies that transactions only target the assigned region. For the SM, no access control is performed. The OS is allowed to access all regions except the ones specified in registers of the arbiter. The firmware is allowed to access its own and the OS regions which is why a static ID needs to be assigned to the firmware.
Peripheral arbiter. We add two registers per peripheral to the arbiter of the peripheral bus. One covers the MMIO region of the peripheral, and the other 32-bit register contains a bitmap that defines read and write permissions for every enclave.
DMA port. We add a register at every port which connects a DMA device. In CURE, a DMA device is exclusively assigned to a single enclave at one point in time. In our prototype, a DMA device accesses the main memory but not other peripherals. If specific use cases, e.g. PCI peer-to-peer transactions [67], must be supported, additional registers need to be added to specify multiple allowed memory regions. Together with the peripheral arbiter, this fulfills FR.2.

6.2.3 L2 Cache Partitioning

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

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

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

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

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

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

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

7 Security Considerations

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

7.1 Hardware Security Primitives (`SR.2`)

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

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

7.2 Enclave Protection (SR.1)

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

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

7.3 Side-channel Attack Resilience (SR.4)

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

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

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

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

8 Evaluation

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

8.1 System Modifications

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

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

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

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

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

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

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

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

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

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

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

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

8.2 Performance Evaluation

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

8.2.1 Microbenchmarks

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

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

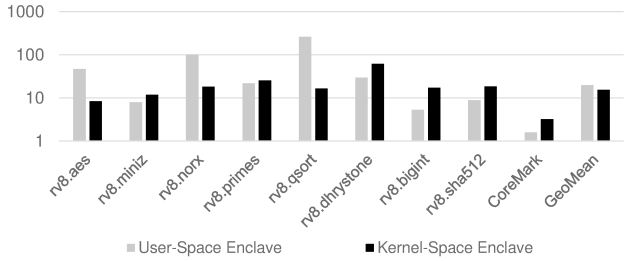


Figure 7: CURE performance overhead (in percent) on macro benchmarks `rv8` and `CoreMark` relative to a normal process.

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

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

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

8.2.2 Macrobenchmarks

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

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

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

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

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

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

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

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

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

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

9 Related Work

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

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

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

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

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

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

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

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

Table 6: Comparison of major TEE architectures with respect to provided enclave types, dyn. cache-side channel and controlled-side channel resilience, and enclave-to-peripheral binding, i.e., MMIO/DMA protection with exclusive enclave assignment. ● indicates full support, ◐ for support with limitations, ○ for no support, * if resilience can only be achieved through extensions.

predefined enclave. Moreover, in contrast to CURE, Keystone does not support enclave-to-peripheral binding.

10 Conclusion

We presented CURE, a novel TEE architecture which provides strongly-isolated enclaves that can be adapted to the functionality and security requirements of the sensitive services which they protect. CURE offers different types of enclaves, ranging from sub-space enclaves, over user-space enclaves, to self-sustained kernel-space enclaves which can execute privileged software. CURE's protection mechanisms are based on new hardware security primitives on the system bus, the shared cache and the CPU. We instantiate CURE on a RISC-V system. The evaluation of our prototype indicates minimal hardware overhead for the security primitives and a moderate overall performance overhead.

Acknowledgments

We thank our anonymous reviewers for their valuable and constructive feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297. Moreover, this project has received funding from Huawei within the OpenS3 lab.

References

[1] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *NDSS*, 2019.

[2] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[3] ARM Limited. Security technology: building a secure system using TrustZone technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2008.

[4] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *TOCS*, 33(3):1–26, 2015.

[6] I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In *CRYPTO*, pages 131–146. Springer, 2000.

[7] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A. Sadeghi. Dr. sgx: automated and adjustable side-channel protection for sgx using data location randomization. In *ACSAC*, pages 788–800, 2019.

[8] F. Brasser, B. El Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *DAC*, pages 1–6. IEEE, 2015.

[9] F. Brasser, T. Frassetto, K. Riedhammer, A. Sadeghi, T. Schneider, and C. Weinert. Voiceguard: Secure and private speech processing. In *Interspeech*, pages 1303–1307, 2018.

[10] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *WOOT*, 2017.

[12] C. Canella, D. Genkin, L. Giner, D. Gruss, et al. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, pages 769–784, 2019.

[13] S. Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *ASPLOS*, volume 13, pages 253–264, 2013.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *EuroS&P*, pages 142–157. IEEE, 2019.

[15] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Asia CCS*, pages 7–18. ACM, 2017.

[16] H. D. Chirammal, P. Mukhedkar, and A. Vettathu. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.

[17] D. Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[18] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *USENIX ATC*, pages 565–578, 2016.

[19] K. Choi, K. Toh, and H. Byun. Realtime training on mobile devices for face recognition applications. *Pattern recognition*, 44(2):386–400, 2011.

[20] J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar. Seca: security-enhanced communication architecture. In *CASES*, pages 78–89. ACM, 2005.

[21] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2019.

[22] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[23] P. Cotret, J. Crenne, G. Gogniat, and J. Diguet. Bus-based mpsoc security through communication protection: A latency-efficient alternative. In *FCCM*, pages 200–207. IEEE, 2012.

[24] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, pages 463–478, 2013.

[25] G. Dessouky, T. Frassetto, and A. Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security*, 2020.

[26] EMBC. Coremark. https://www.eembc.org/coremark/, 2019.

[27] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*, pages 287–305. ACM, 2017.

[28] RISC-V Foundation. The risc-v instruction set manual, volume ii: Privileged architecture. https://riscv.org/specifications/privileged-isa/, 2019.

[29] RISC-V Foundation. Risc-v proxy kernel and boot loader. https://github.com/riscv/riscv-pk, 2019.

[30] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi. Imix: In-process memory isolation extension. In *USENIX Security*, pages 83–97, 2018.

[31] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *USENIX Security*, pages 955–972, 2018.

[32] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *MobiSys*, pages 488–501. ACM, 2017.

[33] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing arm trustzone. In *USENIX Security)*, 2017.

[34] Advanced Micro Devices Inc. Amd64 architecture programmer's manual volume 2: System programming. https://www.amd.com/system/files/TechDocs/24593.pdf, 2019.

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

[36] G. Irazoqui, T. Eisenbarth, and B. Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing and its application to aes. In *S&P*, pages 591–604. IEEE, 2015.

[37] N. P. Jouppi, C. Young, N. Patil, and D. Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, 2018.

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

[39] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *DAC*, page 72. ACM, 2016.

[40] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[41] C. King. stress-ng. https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html, 2019.

[42] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *MICRO*, pages 974–987. IEEE, 2018.

[43] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[44] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, et al. sel4: Formal verification of an os kernel. In *SOSP*, pages 207–220. ACM, 2009.

[45] P. Kocher, J. Horn, A. Fogh, D. Genkin, et al. Spectre attacks: Exploiting speculative execution. In *S&P*, pages 1–19. IEEE, 2019.

[46] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.

[47] P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *EuroSys*, page 10. ACM, 2014.

[48] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović. Keystone: A framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[49] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *S&P*, pages 19–33. IEEE, 2014.

[50] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[51] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *USENIX Security*, pages 1257–1272, 2019.

[52] LibTom. Libtomcrypt. https://www.libtom.net/LibTomCrypt/, 2019.

[53] ARM Limited. Trusted board boot requirements client (tbbr-client) armv8-a. https://static.docs.arm.com/den0006/d/DEN0006D_Trusted_Board_Boot_Requirements.pdf?_ga=2.193628069.980937939.1583698138-225494643.1545056698, 2018.

[54] ARM Limited. Amba® axi and ace protocol specification. https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf, 2019.

[55] Arm Limited. Arm® architecture reference manual. https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf, 2019.

[56] ARM Limited. Arm platform security architecture trusted boot and firmware update. https://pages.arm.com/rs/312-SAX-488/images/DEN0072-PSA_TBFU_1.0-bet1.pdf, 2019.

[57] Linaro. Op-tee. https://www.op-tee.org/.

[58] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, pages 406–418. IEEE, 2016.

[59] F. Liu and R. B. Lee. Random fill cache architecture. In *MICRO*, pages 203–215. IEEE, 2014.

[60] F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *MICRO*, 36(5):8–16, 2016.

[61] John M. Intel software guard extensions remote attestation end-to-end example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example, 2018.

[62] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[63] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *NDSS*, 2019.

[64] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*. ACM, 2013.

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

[66] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, 2013.

[67] NVIDIA. Developing a linux kernel module using gpudirect rdma. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2019.

[68] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.

[69] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *USENIX ATC*, 2018.

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

[71] Orson P. ed25519. https://github.com/orlp/ed25519, 2019.

[72] R. D. Pietro, F. Lombardi, and A. Villani. Cuda leaks: a detailed hack for cuda and a (partial) fix. *TECS*, 15(1):15, 2016.

[73] M. Portnoy. *Virtualization essentials*, volume 19. John Wiley & Sons, 2012.

[74] M. K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787. IEEE, 2018.

[75] RV-8. Rv8-bench. https://github.com/rv8-io/rv8-bench, 2019.

[76] F. L. Sang, V. Nicomette, and Y. Deswarte. I/o attacks in intel pc-based architectures and countermeasures. In *SysSec Workshop*, pages 19–26. IEEE, 2011.

[77] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *USENIX Security*, pages 1357–1374, 2017.

[78] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019.

[79] M. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[80] SiFive. Sifive tilelink specification. https://sifive.cdn.prismic.io/sifive%2F57f93ecf-2c42-46f7-9818-bcdd7d39400a_tilelink-spec-1.7.1.pdf, 2018.

[81] SiFive. Sifive block inclusive cache. https://github.com/sifive/block-inclusivecache-sifive, 2019.

[82] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *S&P*, pages 1–17. IEEE, 2016.

[83] M. Sonka, V. Hlavac, and R. Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[84] D. Steinkraus, I. Buck, and P. Simard. Using gpus for machine learning algorithms. In *ICDAR*, pages 1115–1120. IEEE, 2005.

[85] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *DSN*, 2015.

[86] A. Tang, S. Sethumadhavan, and S. Stolfo. Clkscrew: exposing the perils of security-oblivious energy management. In *USENIX Security*, pages 1057–1074, 2017.

[87] C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *USENIX ATC*, pages 645–658, 2017.

[88] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *USENIX Security*, pages 1221–1238, 2019.

[89] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[90] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[91] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *CCS*, pages 178–195. ACM, 2018.

[92] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017.

[93] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. *S&P*, 2019.

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

[95] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. Codoms: Protecting software with code-centric memory domains. In *ISCA*, pages 469–480. IEEE, 2014.

[96] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *USENIX OSDI 18*, pages 681–696, 2018.

[97] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *DAC*, pages 1–6. ACM, 2016.

[98] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A. Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[99] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. Scattercache: thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019.

[100] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *ISCA*, pages 457–468. IEEE, 2014.

[101] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, pages 640–656. IEEE, 2015.

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

[103] Google Projekt Zero. Trust issues: Exploiting trustzone tees. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017.

[104] Google Projekt Zero. Cve-2018-17182. https://bugs.chromium.org/p/project-zero/issues/detail?id=1664, 2018.

[105] Google Projekt Zero. Xnu: copy-on-write behavior bypass via mount of user-owned filesystem image. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2018.

[106] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *CCS*, pages 1723–1740. ACM, 2019.

[107] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu. Vulnerable gpu memory management: towards recovering raw data from gpu. *Proceedings on Privacy Enhancing Technologies*, 2017(2):57–73, 2017.

I

HARDFAILS: INSIGHTS INTO SOFTWARE-EXPLOITABLE HARDWARE BUGS

[39] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HardFails: Insights into Software-Exploitable Hardware Bugs. In USENIX Security. USENIX Association, 2019. Core Rank A*.

HardFails: Insights into Software-Exploitable Hardware Bugs

Ghada Dessouky and David Gens, ***Technische Universität Darmstadt;*** **Patrick Haney and Garrett Persyn,** ***Texas A&M University;*** **Arun Kanuparthi, Hareesh Khattri, and Jason M. Fung,** ***Intel Corporation;*** **Ahmad-Reza Sadeghi,** ***Technische Universität Darmstadt;*** **Jeyavijayan Rajendran,** ***Texas A&M University***

https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

This paper is included in the Proceedings of the 28th USENIX Security Symposium.

August 14–16, 2019 • Santa Clara, CA, USA

978-1-939133-06-9

Open access to the Proceedings of the 28th USENIX Security Symposium is sponsored by USENIX.

HardFails: Insights into Software-Exploitable Hardware Bugs

Ghada Dessouky[†], David Gens[†], Patrick Haney[*], Garrett Persyn[*], Arun Kanuparthi[°], Hareesh Khattri[°], Jason M. Fung[°], Ahmad-Reza Sadeghi[†], Jeyavijayan Rajendran[*]
[†]*Technische Universität Darmstadt, Germany.* [*]*Texas A&M University, College Station, USA.*
[°]*Intel Corporation, Hillsboro, OR, USA.*
ghada.dessouky@trust.tu-darmstadt.de, david.gens@trust.tu-darmstadt.de,
prh537@tamu.edu, gpersyn@tamu.edu, arun.kanuparthi@intel.com,
hareesh.khattri@intel.com, jason.m.fung@intel.com,
ahmad.sadeghi@trust.tu-darmstadt.de, jv.rajendran@tamu.edu

Abstract

Modern computer systems are becoming faster, more efficient, and increasingly interconnected with each generation. Thus, these platforms grow more complex, with new features continually introducing the possibility of new bugs. Although the semiconductor industry employs a combination of different verification techniques to ensure the security of System-on-Chip (SoC) designs, a growing number of increasingly sophisticated attacks are starting to leverage *cross-layer bugs*. These attacks leverage subtle interactions between hardware and software, as recently demonstrated through a series of real-world exploits that affected all major hardware vendors.

In this paper, we take a deep dive into microarchitectural security from a hardware designer's perspective by reviewing state-of-the-art approaches used to detect hardware vulnerabilities at design time. We show that a protection gap currently exists, leaving chip designs vulnerable to software-based attacks that can exploit these hardware vulnerabilities. Inspired by real-world vulnerabilities and insights from our industry collaborator (a leading chip manufacturer), we construct the first representative testbed of real-world software-exploitable RTL bugs based on RISC-V SoCs. Patching these bugs may not always be possible and can potentially result in a product recall. Based on our testbed, we conduct two extensive case studies to analyze the effectiveness of state-of-the-art security verification approaches and identify specific classes of vulnerabilities, which we call *HardFails*, which these approaches fail to detect. Through our work, we focus the spotlight on specific limitations of these approaches to propel future research in these directions. We envision our RISC-V testbed of RTL bugs providing a rich exploratory ground for future research in hardware security verification and contributing to the open-source hardware landscape.

1 Introduction

The divide between hardware and software security research is starting to take its toll, as we witness increasingly sophisticated attacks that combine software and hardware bugs to exploit computing platforms at runtime [20, 23, 36, 43, 45, 64, 69, 72, 74]. These cross-layer attacks disrupt traditional threat models, which assume either hardware-only or software-only adversaries. Such attacks may provoke physical effects to induce hardware faults or trigger unintended microarchitectural states. They can make these effects visible to software adversaries, enabling them to exploit these hardware vulnerabilities remotely. The affected targets range from low-end embedded devices to complex servers, that are hardened with advanced defenses, such as data-execution prevention, supervisor-mode execution prevention, and control-flow integrity.

Hardware vulnerabilities. Cross-layer attacks circumvent many existing security mechanisms [20, 23, 43, 45, 64, 69, 72, 74], that focus on mitigating attacks exploiting software vulnerabilities. Moreover, hardware-security extensions are not designed to tackle hardware vulnerabilities. Their implementation remains vulnerable to potentially undetected hardware bugs committed at design-time. In fact, deployed extensions such as SGX [31] and TrustZone [3] have been targets of successful cross-layer attacks [69, 72]. Research projects such as Sanctum [18], Sanctuary [8], or Keystone [39] are also not designed to ensure security at the hardware implementation level. Hardware vulnerabilities can occur due to: (a) incorrect or ambiguous security specifications, (b) incorrect design, (c) flawed implementation of the design, or (d) a combination thereof. Hardware implementation bugs are introduced either through human error or faulty translation of the design in gate-level synthesis.

SoC designs are typically implemented at register-transfer level (RTL) by engineers using hardware description languages (HDLs), such as Verilog and VHDL, which are *synthesized* into a lower-level representation using automated tools. Just like software programmers introduce bugs to the high-level code, hardware engineers may accidentally introduce bugs to the RTL code. While software errors typically cause a crash which triggers various fallback routines to ensure the safety and security of other programs running on the platform, no such safety net exists for hardware bugs. Thus, even mi-

nor glitches in the implementation of a module within the processor can compromise the SoC security objectives and result in persistent/permanent denial of service, IP leakage, or exposure of assets to untrusted entities.

Detecting hardware security bugs. The semiconductor industry makes extensive use of a variety of techniques, such as simulation, emulation, and formal verification to detect such bugs. Examples of industry-standard tools include Incisive [10], Solidify [5], Questa Simulation and Questa Formal [44], OneSpin 360 [66], and JasperGold [11]. These were originally designed for *functional verification* with security-specific verification incorporated into them later.

While a rich body of knowledge exists within the software community (e.g., regarding software exploitation and techniques to automatically detect software vulnerabilities [38, 46]), security-focused HDL analysis is currently lagging behind [35, 57]. Hence, the industry has recently adopted a *security development lifecycle* (SDL) for hardware [68] — inspired by software practices [26]. This process combines different techniques and tools, such as RTL manual code audits, assertion-based testing, dynamic simulation, and automated security verification. However, the recent outbreak of *cross-layer attacks* [20, 23, 37, 43, 45, 47, 48, 49, 51, 52, 53, 64, 69, 74] poses a spectrum of difficult challenges for these security verification techniques, because they exploit complex and subtle inter-dependencies between hardware and software. Existing verification techniques are fundamentally limited in modeling and verifying these interactions. Moreover, they also do not scale with the size and complexity of real-world SoC designs.

Goals and Contributions. In this paper, we show that current hardware security verification techniques are fundamentally limited. We provide a wide range of results using a comprehensive test harness, encompassing different types of hardware vulnerabilities commonly found in real-world platforms. To that end, we conducted two case studies to systematically and qualitatively assess existing verification techniques with respect to detecting RTL bugs. Together with our industry partners, we compiled a list of 31 RTL bugs based on public Common Vulnerabilities and Exposures (CVEs) [37, 43, 50, 54, 55] and real-world errata [25]. We injected bugs into two open-source RISC-V-based SoC designs, which we will open-source after publication.

We organized an international public hardware security competition, Hack@DAC, where 54 teams of researchers competed for three months to find these bugs. While a number of bugs could not be detected by any of the teams, several participants also reported *new* vulnerabilities of which we had no prior knowledge. The teams used manual RTL inspection and simulation techniques to detect the bugs. In industry, these are usually complemented by automated tool-based and formal verification approaches. Thus, our second case study focused on two state-of-the-art formal verification tools: the first deploys formal verification to perform exhaustive and complete verification of a hardware design, while the second leverages formal verification and path sensitization to check for illegal data flows and fault tolerance.

Our second case study revealed that certain properties of RTL bugs pose challenges for state-of-the-art verification techniques with respect to black-box abstraction, timing flow, and non-register states. This causes security bugs in the RTL of real-world SoCs to slip through the verification process. Our results from the two case studies indicate that particular classes of hardware bugs entirely evade detection—even when complementing systematic tool-based verification approaches with manual inspection. RTL bugs arising from complex and cross-modular interactions in SoCs render these bugs extremely difficult to detect in practice. Furthermore, such bugs are exploitable from software, and thus can compromise the entire platform. We call such bugs **HardFails**.

To the best of our knowledge, this is the first work to provide a systematic and in-depth analysis of state-of-the-art hardware verification approaches for security-relevant RTL bugs. Our findings shed light on the capacity of these tools and demonstrate reproducibly how bugs can slip through current hardware security verification processes. Being also software-exploitable, these bugs pose an immense security threat to SoCs. Through our work, we highlight why further research is required to improve state-of-the-art security verification of hardware. To summarize, our main contributions are:

- **Systematic evaluation and case studies:** We compile a comprehensive test harness of real-world RTL bugs, on which we base our two case studies: (1) Hack@DAC'18, in which 54 independent teams of researchers competed worldwide over three months to find these bugs using manual RTL inspection and simulation techniques, and (2) an investigation of the bugs using industry-leading formal verification tools that are representative of the current state of the art. Our results show that particular classes of bugs entirely evade detection, despite combining both tool-based security verification approaches and manual analysis.
- **Stealthy hardware bugs:** We identify *HardFails* as RTL bugs that are distinctly challenging to detect using industry-standard security verification techniques. We explain the fundamental limitations of these techniques in detail using concrete examples.
- **Open-sourcing:** We will open-source our bugs testbed at publication to the community.

2 SoC Verification Processes and Pitfalls

Similar to the Security Development Lifecycle (SDL) deployed by software companies [26], semiconductor companies [15, 35, 40] have recently adapted SDL for hardware design [57]. We describe next the conventional SDL process for hardware and the challenges thereof.

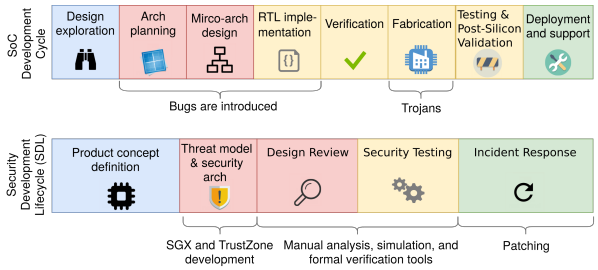


FIGURE 1: Typical Security Development Lifecycle (SDL) process followed by semiconductor companies.

2.1 The Security Development Lifecycle (SDL) for Hardware

SDL is conducted concurrently with the conventional hardware development lifecycle [68], as shown in Figure 1. The top half of Figure 1 shows the hardware development lifecycle. It begins with design exploration followed by defining the specifications of the product architecture. After the architecture specification, the microarchitecture is designed and implemented in RTL. Concurrently, pre-silicon verification efforts are conducted until tape-out to detect and fix all functional bugs that do not meet the *functional specification*. After tape-out and fabrication, iterations of post-silicon validation, functional testing, and tape-out "spins" begin. This cycle is repeated until no defects are found and all quality requirements are met. Only then does the chip enter mass production and is shipped out. Any issues found later in-field are debugged, and the chip is then either patched if possible or recalled.

After architectural features are finalized, a security assessment is performed, shown in the bottom half of Figure 1. The adversary model and the security objectives are compiled in the *security specification*. This typically entails a list of assets, entry points to access these assets, and the adversary capabilities and architectural security objectives to mitigate these threats. These are translated into microarchitectural security specifications, including security test cases (both positive and negative). After implementation, pre-silicon security verification is conducted using dynamic verification (i.e., simulation and emulation), formal verification, and manual RTL reviews. The chip is not signed off for tape-out until all security specifications are met. After tape-out and fabrication, post-silicon security verification commences. The identified security bugs in both pre-silicon and post-silicon phases are rated for severity using the industry-standard scoring systems such as the Common Vulnerability Scoring System (CVSS) [30] and promptly fixed. Incident response teams handle issues in shipped products and provide patches, if possible.

2.2 Challenges with SDL

Despite multiple tools and security validation techniques used by industry to conduct SDL, it remains a highly challenging, tedious, and complex process even for industry experts. Existing techniques largely rely on human expertise to define the security test cases and run the tests. The correct *security specifications* must be exhaustively anticipated, identified, and accurately and adequately expressed using security properties that can be captured and verified by the tools. We discuss these challenges further in Section 7.

Besides the specifications, the techniques and tools themselves are not scalable and are less effective in capturing subtle semantics that are relevant to many vulnerabilities, which is the focus of this work. We elaborate next on the limitations of state-of-the-art hardware security verification tools commonly used by industry. To investigate the capabilities of these tools, we then construct a comprehensive test-harness of real-world RTL vulnerabilities.

3 Assessing Hardware Security Verification

In this section, we focus on why the verification of the security properties of modern hardware is challenging and provide requirements for assessing existing verification techniques under realistic conditions. First, we describe how these verification techniques fall short. Second, we provide a list of common and realistic classes of hardware bugs, which we use to construct a test harness for assessing the effectiveness of these verification techniques. Third, we discuss how these bugs relate to the common security goals of a chip.

3.1 Limitations of Automated Verification

Modern hardware designs are highly complex and incorporate hundreds of in-house and third-party Intellectual Property (IP) components. This creates room for vulnerabilities to be introduced in the inter-modular interactions of the design hierarchy. Multi-core architectures typically have an intricate interconnect fabric between individual cores (utilizing complex communication protocols), multi-level cache controllers with shared un-core and private on-core caches, memory and interrupt controllers, and debug and I/O interfaces.

For each core, these components contain logical modules such as fetch and decode stages, an instruction scheduler, individual execution units, branch prediction, instruction and data caches, the memory subsystem, re-order buffers, and queues. These are implemented and connected using individual RTL modules. The average size of each module is several hundred lines of code (LOC). Thus, real-world SoCs can easily approach 100,000 lines of RTL code, and some designs may even have millions of LOC. Automatically verifying, at the RTL level, the respective interconnections and checking them against security specifications raises a number of fundamental challenges for the state-of-the-art approaches. These are described below.

L-1: Cross-modular effects. Hardware modules are interconnected in a highly hierarchical design with multiple inter-

dependencies. Thus, an RTL bug located in an individual module may trigger a vulnerability in intra- and inter-modular information flows spanning multiple complex modules. Pinpointing the bug requires analyzing these flows across the relevant modules, which is highly cumbersome and unreliable to achieve by manual inspection. It also pushes formal verification techniques to their limits, which work by modeling and analyzing all the RTL modules of the design to verify whether design specifications (expressed using security property assertions, invariants and disallowed information flows) and implementation match.

Detecting such vulnerabilities requires loading the RTL code of all the relevant modules into the tools to model and analyze the entire state space, thus driving them quickly into *state explosion* due to the underlying modeling algorithms [16, 21]. Alleviating this by providing additional computational resources and time is not scalable as the complexity of SoCs continues to increase. Selective "black-box" abstraction of some of the modules, state space constraining, and bounded-model checking are often used. However, they do not eliminate the fundamental problem and rely on interactive human expertise. Erroneously applying them may introduce false negatives, leading to missed vulnerabilities.

L-2: Timing-flow gap. Current industry-standard techniques are limited in capturing and verifying security properties related to timing flow (in terms of clock cycle latency). This leads to vast sources of information leakage due to software-exploitable timing channels (Section 8). A timing flow exists between the circuit's input and output when the number of clock cycles required for the generation of the output depends on input values or the current memory/register state. This can be exploited to leak sensitive information when the timing variation is discernible by an adversary and can be used to infer inputs or memory states. This is especially problematic for information flows and resource sharing across different privilege levels. This timing variation should remain indistinguishable in the RTL, or should not be measurable from the software. However, current industry-standard security verification techniques focus exclusively on the functional information flow of the logic and fail to model the associated timing flow. The complexity of timing-related security issues is aggravated when the timing flow along a logic path spans multiple modules and involves various inter-dependencies.

L-3: Cache-state gap. State-of-the-art verification techniques only model and analyze the *architectural state* of a processor by exclusively focusing on the state of registers. However, they do not support analysis of non-register states, such as caches, thus completely discarding modern processors' highly complex microarchitecture and diverse hierarchy of caches. This can lead to severe security vulnerabilities arising due to state changes that are unaccounted for, e.g., the changing state of shared cache resources across multiple privilege levels. Caches represent a state that is influenced directly or indirectly by many control-path signals and can generate security vulnerabilities in their interactions, such as illegal information leakages across different privilege levels. Identifying RTL bugs that trigger such vulnerabilities is beyond the capabilities of existing techniques.

L-4: Hardware-software interactions. Some RTL bugs remain indiscernible to hardware security verification techniques because they are not explicitly vulnerable unless triggered by the software. For instance, although many SoC access control policies are directly implemented in hardware, some are programmable by the overlying firmware to allow for post-silicon flexibility. Hence, reasoning on whether an RTL bug exists is inconclusive when considering the hardware RTL in isolation. These vulnerabilities would only materialize when the hardware-software interactions are considered, and existing techniques do not handle such interactions.

3.2 Constructing Real-World RTL Bugs

To systematically assess the state of the art in hardware security verification with respect to the limitations described above, we construct a test harness by implementing a large number of RTL bugs in RISC-V SoC designs (cf. Table 1). To the best of our knowledge, we are the first to compile and showcase such a collection of hardware bugs. Together with our co-authors at Intel, we base our selection and construction of bugs on a solid representative spectrum of real-world CVEs [47, 48, 49, 51, 52, 53] as shown in Table 1. For instance, bug #22 was inspired by a recent security vulnerability in the Boot ROM of video gaming mobile processors [56], which allowed an attacker to bring the device into BootROM Recovery Mode (RCM) via USB access. This buffer overflow vulnerability affected many millions of devices and is popularly used to hack a popular video gaming console[1].

We extensively researched CVEs that are based on software-exploitable hardware and firmware bugs and classified them into different categories depending on the weaknesses they represent and the modules they impact. We reproduced them by constructing representative bugs in the RTL and demonstrated their software exploitability and severity by crafting a real-world software exploit based on one of these bugs in Appendix D. Other bugs were constructed with our collaborating hardware security professionals, inspired by bugs that they have previously encountered and patched during the pre-silicon phase, which thus never escalated into CVEs. The chosen bugs were implemented to achieve coverage of different security-relevant modules of the SoC.

Since industry-standard processors are based on proprietary RTL implementations, we mimic the CVEs by reproducing and injecting them into the RTL of widely-used RISC-V SoCs. We also investigate more complex microarchitecture features of another RISC-V SoC and discover vulnerabilities already existing in its RTL (Section 4). These RTL bugs manifest as:

[1] https://github.com/Cease-and-DeSwitch/fusee-launcher

- **Incorrect assignment bugs** due to variables, registers, and parameters being assigned incorrect literal values, incorrectly connected or left floating unintended.
- **Timing bugs** resulting from timing flow issues and incorrect behavior relevant to clock signaling such as information leakage.
- **Incorrect case statement bugs** in the finite state machine (FSM) models such as incorrect or incomplete selection criteria, or incorrect behavior within a case.
- **Incorrect if-else conditional bugs** due to incorrect boolean conditions or incorrect behavior described within either branch.
- **Specification bugs** due to a mismatch between a specified property and its actual implementation or poorly specified / under-specified behavior.

These seemingly minor RTL coding errors may constitute security vulnerabilities, some of which are very difficult to detect during verification. This is because of their interconnection and interaction with the surrounding logic that affects the complexity of the subtle side effects they generate in their manifestation. Some of these RTL bugs may be patched by modifying parts of the software stack that use the hardware (e.g., using firmware/microcode updates) to circumvent them and mitigate specific exploits. However, since RTL is usually compiled into hardwired integrated circuitry logic, the underlying bugs cannot, in principle, be patched after production.

The limited capabilities of current detection approaches in modeling hardware designs and formulating and capturing relevant security assertions raise challenges for detecting some of these vulnerabilities, which we investigate in depth in this work. We describe next the adversary model we assume for our vulnerabilities and our investigation.

3.3 Adversary Model

In our work, we investigate microarchitectural details at the RTL level. However, all hardware vendors keep their proprietary industry designs and implementations closed. Hence, we use an open-source SoC based on the popular open-source RISC-V [73] architecture as our platform. RISC-V supports a wide range of possible configurations with many standard features that are also available in modern processor designs, such as privilege level separation, virtual memory, and multi-threading, as well as optimization features such as configurable branch prediction and out-of-order execution.

RISC-V RTL is freely available and open to inspection and modification. While this is not necessarily the case for industry-leading chip designs, an adversary might be able to reverse engineer or disclose/steal parts of the chip using existing tools[2][3]. Hence, we consider a strong adversary that can also inspect the RTL code.

In particular, we make the following assumptions:

- **Hardware Vulnerability:** The attacker has knowledge of a vulnerability in the hardware design of the SoC (i.e., at the RTL level) and can trigger the bug from software.
- **User Access:** The attacker has complete control over a user-space process, and thus can issue unprivileged instructions and system calls in the basic RISC-V architecture.
- **Secure Software:** Software vulnerabilities and resulting attacks, such as code-reuse [65] and data-only attacks [27] against the software stack, are orthogonal to the problem of cross-layer bugs. Thus, we assume all platform software is protected by defenses such as control-flow integrity [1] and data-flow integrity [13], or is formally verified.

The goal of an adversary is to leverage the vulnerability on the chip to provoke unintended functionality, e.g., access to protected memory locations, code execution with elevated privileges, breaking the isolation of other processes running on the platform, or permanently denying services. RTL bugs in certain hardware modules might only be exploitable with physical access to the victim device, for instance, bugs in debug interfaces. However, other bugs are software-exploitable, and thus have a higher impact in practice. Hence, we focus on software-exploitable RTL vulnerabilities, such as the exploit showcased in Appendix D. Persistent denial of service (PDoS) attacks that require exclusive physical access are out of scope. JTAG attacks, though they require physical access, are still in scope as the end user may be the attacker and might attempt to unlock the device to steal manufacturer secrets. Furthermore, exploiting the JTAG interface often requires a combination of both physical access and privilege escalation by means of a software exploit to enable the JTAG interface. We also note that an adversary with unprivileged access is a realistic model for real-world SoCs: Many platforms provide services to other devices over the local network or even over the internet. Thus, the attacker can obtain some limited software access to the platform already, e.g., through a webserver or an RPC interface. Furthermore, we emphasize that this work focuses only on tools and techniques used to detect bugs before tape-out.

4 HardFails: Hardware Security Bugs

In light of the limitations of state-of-the-art verification tools (Section 3.1), we constructed a testbed of real-world RTL bugs (Section 3.2) and conducted two extensive case studies on their detection (described next in Sections 5 and 6). Based on our findings, we have identified particular classes of hardware bugs that exhibit properties that render them more challenging to detect with state-of-the-art techniques. We call these HardFails. We now describe different types of these HardFails encountered during our analysis of two RISC-V SoCs, Ariane [59] and PULPissimo [61]. In Section 5.3, we describe the actual bugs we instantiated for our case studies.

Ariane is a 6-stage in-order RISC-V CPU that implements the RISC-V draft privilege specification and can run Linux OS. It has a memory management unit (MMU) consisting of

[2] https://www.chipworks.com/
[3] http://www.degate.org/

TABLE 1: Detection results for bugs in PULPissimo SoC based on formal verification (**SPV** and **FPV**, i.e., JasperGold Security Path Verification and Formal Property Verification) and our hardware security competition (**M&S**, i.e., manual inspection and simulation). Check and cross marks indicate detected and undetected bugs, respectively. Bugs marked **inserted** were injected by our team and based on the listed CVEs, while bugs marked **native** were already present in the SoC and discovered by the participants during Hack@DAC. **LOC** denotes the number of lines of code, and **states** denotes the total number of logic states for the modules needed to attempt to detect this bug.

#	Bug	Type	SPV	FPV	M&S	Modules	LOC	# States
1	Address range overlap between peripherals SPI Master and SoC	Inserted (CVE-2018-12206 / CVE-2019-6260 / CVE-2018-8933)	✓	✓	✓	91	6685	1.5×10^{20}
2	Addresses for L2 memory is out of the specified range.	Native	✓	✓	✓	43	6746	3.5×10^{13}
3	Processor assigns privilege level of execution incorrectly from CSR.	Native	✗	✓	✓	2	1186	2.1×10^{96}
4	Register that controls GPIO lock can be written to with software.	Inserted (CVE-2017-18293)	✓	✓	✗	2	1186	2.1×10^{96}
5	Reset clears the GPIO lock control register.	Inserted (CVE-2017-18293)	✓	✓	✗	2	408	1
6	Incorrect address range for APB allows memory aliasing.	Inserted (CVE-2018-12206 / CVE-2019-6260)	✓	✓	✗	1	110	2
7	AXI address decoder ignores errors.	Inserted (CVE-2018-4850)	✗	✓	✗	1	227	2
8	Address range overlap between GPIO, SPI, and SoC control peripherals.	Inserted (CVE-2018-12206 / (CVE-2017-5704)	✓	✓	✓	68	14635	9.4×10^{21}
9	Incorrect password checking logic in debug unit.	Inserted (CVE-2018-8870)	✗	✓	✗	4	436	1
10	Advanced debug unit only checks 31 of the 32 bits of the password.	Inserted (CVE-2017-18347 / CVE-2017-7564)	✗	✓	✗	4	436	16
11	Able to access debug register when in halt mode.	Native (CVE-2017-18347 /	✗	✓	✓	2	887	1
12	Password check for the debug unit does not reset after successful check.	Inserted (CVE-2017-7564)	✗	✓	✓	4	436	16
13	Faulty decoder state machine logic in RISC-V core results in a hang.	Native	✗	✓	✓	2	1119	32
14	Incomplete case statement in ALU can cause unpredictable behavior.	Native	✗	✓	✓	2	1152	4
15	Faulty logic in the RTC causing inaccurate time calculation for security-critical flows, e.g., DRM.	Native	✗	✓	✗	1	191	1
16	Reset for the advanced debug unit not operational.	Inserted (CVE-2017-18347)	✗	✗	✓	4	436	16
17	Memory-mapped register file allows code injection.	Native	✗	✗	✓	1	134	1
18	Non-functioning cryptography module causes DOS.	Inserted	✗	✗	✗	24	2651	1
19	Insecure hash function in the cryptography module.	Inserted (CVE-2018-1751)	✗	✗	✗	24	2651	N/A
20	Cryptographic key for AES stored in unprotected memory.	Inserted (CVE-2018-8933 / CVE-2014-0881 / CVE-2017-5704)	✗	✗	✗	57	8955	1
21	Temperature sensor is muxed with the cryptography modules.	Inserted	✗	✗	✓	1	65	1
22	ROM size is too small preventing execution of security code.	Inserted (CVE-2018-6242 /) CVE-2018-15383)	✗	✗	✓	1	751	N/A
23	Disabled the ability to activate the security-enhanced core.	Inserted (CVE-2018-12206)	✗	✗	✗	1	282	N/A
24	GPIO enable always high.	Inserted (CVE-2018-1959)	✗	✗	✗	1	392	1
25	Unprivileged user-space code can write to the privileged CSR.	Inserted (CVE-2018-7522 / CVE-2017-0352)	✗	✗	✓	1	745	1
26	Advanced debug unit password is hard-coded and set on reset.	Inserted (CVE-2018-8870)	✗	✗	✓	1	406	16
27	Secure mode is not required to write to interrupt registers.	Inserted (CVE-2017-0352)	✗	✗	✓	1	303	1
28	JTAG interface is not password protected.	Native	✗	✗	✓	1	441	1
29	Output of MAC is not erased on reset.	Inserted	✗	✗	✓	1	65	1
30	Supervisor mode signal of a core is floating preventing the use of SMAP.	Native	✗	✗	✓	1	282	1
31	GPIO is able to read/write to instruction and data cache.	Native	✗	✗	✓	1	151	4

data and instruction translation lookaside buffers (TLBs), a hardware page table walker, and a branch prediction unit to enable speculative execution. Figure 4 in Appendix A shows its high-level microarchitecture.

PULPissimo is an SoC based on a simpler RISC-V core with both instruction and data RAM as shown in Figure 2. It provides an Advanced Extensible Interface (AXI) for accessing memory from the core. Peripherals are directly connected to an Advanced Peripheral Bus (APB) which connects them to the AXI through a bridge module. It provides support for autonomous I/O, external interrupt controllers and features a debug unit and an SPI slave.

TLB Page Fault Timing Side Channel (L-1 & L-2). While analyzing the Ariane RTL, we noted a timing side-channel leakage with TLB accesses. TLB page faults due to illegal accesses occur in a different number of clock cycles than page faults that occur due to unmapped memory (we contacted the developers and they acknowledged the vulnerability). This timing disparity in the RTL manifests in the microarchitectural behavior of the processor. Thus, it constitutes a software-visible side channel due to the measurable clock-cycle difference in the two cases. Previous work already demonstrated how this can be exploited by user-space adversaries to probe mapped and unmapped pages and to break randomization-based defenses [24, 29]. Timing flow properties cannot be directly expressed by simple properties or modeled by state-of-the-art verification techniques. Moreover, for this vulnerability, we identify at least seven RTL modules that would need to be modeled, analyzed and verified in combination, namely: *mmu.sv - nbdcache.sv - tlb.sv instantiations - ptw.sv - load_unit.sv - store_unit.sv*. Besides modeling their complex inter- and intra-modular logic flows (L-1), the timing flows need to be modeled to formally prove the absence of this timing channel leakage, which is *not supported* by current industry-standard tools (L-2). Hence, the only alternative is to verify this property by manually inspecting and following the clock cycle transitions across the RTL modules, which is highly cumbersome and error-prone. However, the design must still be analyzed to verify that timing side-channel resilience is implemented correctly and bug-free in the RTL. This only becomes far more complex for real-world industry-standard SoCs. We show the RTL hierarchy of the Ariane core in Figure 5 in Appendix A to illustrate its complexity.

Pre-Fetched Cache State Not Rolled Back (L-1 & L-3). Another issue in Ariane is with the cache state when a system return instruction is executed, where the privilege level of the core is not changed until this instruction is retired. Before retirement, linear fetching (guided by branch prediction) of data and instructions following the unretired system return instruction continues at the current higher system privilege level. Once the instruction is retired, the execution mode of the core is changed to the unprivileged level, but the entries that were pre-fetched into the cache (at the system privilege level) do not get flushed. These shared cache entries are visible to user-space software, thus enabling timing channels between privileged and unprivileged software.

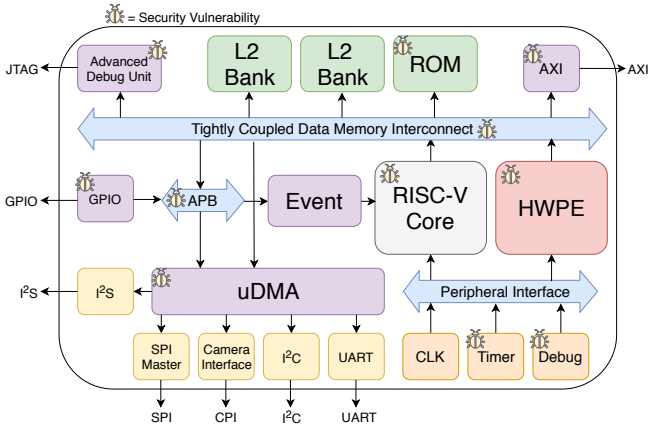


FIGURE 2: Hardware overview of the PULPissimo SoC. Each bug icon indicates the presence of at least one security vulnerability in the module.

Verifying the implementation of all the flush control signals and their behavior in all different states of the processor requires examining at least eight modules: *ariane.sv - controller.sv - frontend.sv - id_stage.sv - icache.sv - fetch_fifo - ariane_pkg.sv - csr_regfile.sv* (see Figure 5). This is complex because it requires identifying and defining all the relevant security properties to be checked across these RTL modules. Since current industry-standard approaches do not support expressive capturing and the verification of cache states, this issue in the RTL can only be found by manual inspection.

Firmware-Configured Memory Ranges (L-4). In PULPissimo, we added peripherals with injected bugs to reproduce bugs from CVEs. We added an AES encryption/decryption engine whose input key is stored and fetched from memory tightly coupled to the processor. The memory address the key is stored in is unknown, and whether it is within the protected memory range or not is inconclusive by observing the RTL alone. In real-world SoCs, the AES key is stored in programmable fuses. During secure boot, the bootloader/firmware senses the fuses and stores the key to memory-mapped registers. The access control filter is then configured to allow only the AES engine access to these registers, thus protecting this memory range. Because the open-source SoC we used did not contain a fuse infrastructure, the key storage was mimicked to be in a register in the Memory-Mapped I/O (MMIO) space.

Although the information flow of the AES key is defined in hardware, its location is controlled by the firmware. Reasoning on whether the information flow is allowed or not using conventional hardware verification approaches is inconclusive when considering the RTL code in isolation.

The vulnerable hardware/firmware interactions cannot be identified unless they are co-verified. Unfortunately, current industry-standard tools do not support this.

Memory Address Range Overlap (L-1 & L-4). PULPissimo provides I/O support to its peripherals by mapping them to different memory address ranges. If an address range overlap bug is committed at design-time or by firmware, this can break access control policies and have critical security consequences, e.g., privilege escalation. We injected an RTL bug where there is address range overlap between the SPI Master Peripheral and the SoC Control Peripheral. This allowed the untrusted SPI Master to access the SoC Control memory address range over the APB bus.

Verifying issues at the SoC interconnect in such complex bus protocols is challenging since too many modules needed to support the interconnect have to be modeled to properly verify their security. This increases the scope and the complexity of potential bugs far beyond just a few modules, as shown in Table 1. Such an effect causes an explosion of the state space since all the possible states have to be modeled accurately to remain sound. Proof kits for accelerated verification of advanced SoC interconnect protocols were introduced to mitigate this for a small number of bus protocols (AMBA3 and AMBA4). However, this requires an add-on to the default software and many protocols are not supported[4].

5 Crowdsourcing Detection

We organized and conducted a capture-the-flag competition, Hack@DAC, in which 54 teams (7 from leading industry vendors and 47 from academia) participated. The objective for the teams was to detect as many RTL bugs as possible from those we injected deliberately in real-world open-source SoC designs (see Table 1). This is designed to mimic real-world bug bounty programs from semiconductor companies [17, 32, 62, 63]. The teams were free to use any techniques: simulation, manual inspection, or formal verification.

5.1 Competition Preparation

RTL of open-source RISC-V SoCs was used as the testbed for Hack@DAC and our investigation. Although these SoCs are less complex than high-end industry proprietary designs, this allows us to feasibly inject (and detect) bugs into less complex RTL. Thus, this represents the best-case results for the verification techniques used during Hack@DAC and our investigation. Moreover, it allows us to open-source and showcase our testbed and bugs to the community.Hack@DAC consisted of two phases: a preliminary Phase 1 and final Phase 2, which featured the RISC-V Pulpino and PULPissimo SoCs, respectively. Phase 1 was conducted remotely over a two-month period. Phase 2 was conducted in an 8-hour time frame co-located with DAC (Design Automation Conference).

For Phase 1, we chose the Pulpino [60] SoC since it was a real-world, yet not an overly complex SoC design for the teams to work with. It features a RISC-V core with instruction and data RAM, an AXI interconnect for accessing memory, with peripherals on an APB accessing the AXI through a bridge module. It also features a boot ROM, a debug unit and a serial peripheral interface (SPI) slave. We inserted security bugs in multiples modules of the SoC, including the AXI, APB, debug unit, GPIO, and bridge.

For Phase 2, we chose the more complex PULPissimo [61] SoC, shown in Figure 2. It additionally supports hardware processing engines, DMA, and more peripherals. This allowed us to extend the SoC with additional security features, making room for additional bugs. Some native security bugs were discovered by the teams and were reported to the SoC designers.

5.2 Competition Objectives

For Hack@DAC, we first implemented additional security features in the SoC, then defined the security objectives and adversary model and accordingly inserted the bugs. Specifying the security goals and the adversary model allows teams to define what constitutes a security bug. Teams had to provide a bug description, location of RTL file, code reference, the security impact, adversary profile, and the proposed mitigation.
Security Features: We added password-based locks on the JTAG modules of both SoCs and access control on certain peripherals. For the Phase-2 SoC, we also added a cryptographic unit implementing multiple cryptographic algorithms. We injected bugs into these features and native features to generate security threats as a result.
Security Goals: We provided the three main security goals for the target SoCs to the teams. Firstly, unprivileged code should not escalate beyond its privilege level. Secondly, the JTAG module should be protected against an adversary with physical access. Finally, the SoCs should thwart software adversaries from launching denial-of-service attacks.

5.3 Overview of Competition Bugs

As described earlier in Section 3.2, the bugs were selected and injected together with our Intel collaborators. They are inspired by their hardware security expertise and real-world CVEs (cf. Table 1) and aim to achieve coverage of different security-relevant components of the SoC. Several participants also reported a number of *native* bugs already present in the SoC that we did not deliberately inject. We describe below some of the most interesting bugs.
UDMA address range overlap: We modified the memory address range of the UDMA so that it overlaps with the master port to the SPI. This bug allows an adversary with access to

[4] http://www.marketwired.com/press-release/jasper-introduces-intelligent-proof-kits-faster-more-accurate-verification-soc-interface-1368721.htm

the UMDA memory to escalate its privileges and modify the SPI memory. This bug is an example of the "Memory Address Range Overlap" HardFail type in Section 4. Other address range configuration bugs (#1, 2, 6 and 8) were also injected in the APB bus for different peripherals.

GPIO errors: The address range of the GPIO memory was erroneously declared. An adversary with GPIO access can escalate its privilege and access the SPI Master and SoC Control. The GPIO enable was rigged to display a fixed erroneous status of '1', which did not give the user a correct display of the actual GPIO status. The GPIO lock control register was made write-accessible by user-space code, and it was flawed to clear at reset. Bugs #4, 5, 24 and 31 are such examples.

Debug/JTAG errors: The password-checking logic in the debug unit was flawed and its state was not being correctly reset after a successful check. We hard-coded the debug unit password, and the JTAG interface was not password protected. Bugs #9, 10, 11, 16, 26, and 28 are such examples.

Untrusted boot ROM: A native bug (bug #22) would allow unprivileged compromise of the boot ROM and potentially the execution of untrusted boot code at a privileged level, thus disclosing sensitive information.

Erroneous AXI finite-state machine: We injected a bug (bug #7) in the AXI address decoder such that, if an error signal is generated on the memory bus while the underlining logic is still handling an outstanding transaction, the next signal to be handled will instead be considered operational by the module unconditionally. This bug can be exploited to cause computational faults in the execution of security-critical code (we showcase how to exploit this vulnerability—which was not detected by all teams—in Appendix D).

Cryptographic unit bugs: We injected bugs in a cryptographic unit that we inserted to trigger denial-of-service, a broken cryptographic implementation, insecure key storage, and disallowed information leakage. Bugs #18, 19, 20, 21, and 29 are such examples.

5.4 Competition Results

Various insights were drawn from the submitted bug reports and results, which are summarized in Table 1.

Analyzing the bug reports: Bug reports submitted by teams revealed which bug types were harder to detect and analyze using existing approaches. We evaluated the submissions and rated them for accuracy and detail, e.g., bug validity, methodology used, and security impact.

Detected bugs: Most teams easily detected two bugs in PULPissimo. The first one is where debug IPs were used when not intended. The second bug was where we declared a local parameter PULP_SEC, which was always set to '1', instead of the intended PULP_SECURE. The former was detected because debugging interfaces represent security-critical regions of the chip. The latter was detected because it indicated intuitively that exploiting this parameter would lead to privilege escalation attacks. The teams reported that they prioritized inspecting security-relevant modules of the SoC, such as the debug interfaces.

Undetected bugs: Many inserted bugs were not detected. One was in the advanced debug unit, where the password bit index register has an overflow (bug #9). This is an example of a security flaw that would be hard to detect by methods other than verification. Moreover, the presence of many bugs within the advanced debug unit password checker further masked this bug. Another bug was the cryptographic unit key storage in unprotected memory (bug #20). The teams could not detect it as they focused on the RTL code in isolation and did not consider HW/FW interactions.

Techniques used by the teams: The teams were free to use any techniques to detect the bugs but most teams eventually relied on manual inspection and simulation.

- **Formal verification:** One team used an open-source formal verification tool (VeriCoq), but they reported little success because these tools (i) did not scale well with the complete SoC and (ii) required expertise to use and define the security properties. Some teams deployed their in-house verification techniques, albeit with little success. They eventually resorted to manual analysis.
- **Assertion-based simulation:** Some teams prepared RTL testbenches and conducted property-based simulations using SystemVerilog assertion statements.
- **Manual inspection:** All teams relied on manual inspection methods since they are the easiest and most accessible and require less expertise than formal verification, especially when working under time constraints. A couple of teams reported prioritizing the inspection of security-critical modules such as debug interfaces.
- **Software-based testing:** One team detected software-exposure and privilege escalation bugs by running C code on the processor and attempting to make arbitrary reads/writes to privileged memory locations. In doing this, they could detect bugs #4, #8, #15, and #17.

Limitations of manual analysis: While manual inspection can detect the widest array of bugs, our analysis of the Hack@DAC results reveals its limitations. Manual analysis is qualitative and difficult to scale to cross-layer and more complex bugs. In Table 1, out of 16 cross-module bugs (spanning more than one module) only 9 were identified using manual inspection. Three of them (#18, #19, and #20) were also undetected by formal verification methods, which is 10% of the bugs in our case studies.

6 Detection Using State-of-The-Art Tools

Our study reveals two results: (1) a number of bugs could not be detected by means of manual auditing and other ad-hoc methods, and (2) the teams were able to find bugs already existing in the SoC which we did not inject and were not

aware of. This prompted us to conduct a second in-house case study to further investigate whether formal verification techniques can be used to detect these bugs. In practice, hardware-security verification engineers use a combination of techniques such as formal verification, simulation, emulation, and manual inspection. Our first case study covered manual inspection, simulation and emulation techniques. Thus, we focused our second case study on assessing the effectiveness of industry-standard formal verification techniques usually used for verifying pre-silicon hardware security.

In real-world security testing (see Section 2), engineers will not have prior knowledge of the specific vulnerabilities they are trying to find. Our goal, however, is to investigate how an industry-standard tool can detect RTL bugs that we deliberately inject in an open-source SoC and have prior knowledge of (see Table 1). Since there is no regulation or explicitly defined standard for hardware-security verification, we focus our investigation on the most popular and de-facto standard formal verification platform used in industry [11]. This platform encompasses a representative suite of different state-of-the-art formal verification techniques for hardware security assurance. As opposed to simulation and emulation techniques, formal verification guarantees to model the state space of the design and formally prove the desired properties. We emphasize that we deliberately fix all other variables involved in the security testing process, in order to focus in a controlled setting on testing the capacity and limitations of the techniques and tools themselves. Thus, our results reflect the effectiveness of tools in a best case where the bug is known a priori. This eliminates the possibility of writing an incorrect security property assertion which fails to detect the bug.

6.1 Detection Methodology

We examined each of the injected bugs and its nature in order to determine which formal technique would be best suited to detect it. We used two formal techniques: Formal Property Verification (FPV) and JasperGold's Security Path Verification (SPV) [12]. They represent the state of the art in hardware security verification and are used widely by the semiconductor industry [4], including Intel.

FPV checks whether a set of security properties, usually specified as SystemVerilog Assertions (SVA), hold true for the given RTL. To describe the assertions correctly, we examined the location of each bug in the RTL and how its behavior is manifested with the surrounding logic and input/output relationships. Once we specified the security properties using *assert*, *assume* and *cover* statements, we determined which RTL modules we need to model to prove these assertions. If a security property is violated, the tool generates a counterexample; this is examined to ensure whether the intended security property is indeed violated or is a false alarm.

SPV detects bugs which specifically involve unauthorized information flow. Such properties cannot be directly captured using SVA/PSL assertions. SPV uses path sensitization techniques to exhaustively and formally check if unauthorized data propagates (through a functional path) from a source to a destination signal. To specify the SPV properties, we identified source signals where the sensitive information was located and destination signals where it should *not* propagate. We then identified the bounding preconditions to constrain the paths the tool searches to alleviate state and time explosion. Similar to FPV, we identified the modules that are required to capture the information flow of interest. This must include source, destination and intermediate modules, as well as modules that generate control signals which interfere with the information flow.

FIGURE 3: Verification results grouped by bug class and the number of bugs in each class detected by Security Path Verification (SPV), Formal Property Verification (FPV) and manual inspection and simulation techniques (M&S).

6.2 Detection Results

Of the 31 bugs we investigated, shown in Table 1, using the formal verification techniques described above, only 15 (48%) were detected. While we attempted to detect all 31 bugs formally, we were able to formulate security properties for only 17 bugs. This indicates that the main challenge with using formal verification tools is identifying and expressing security properties that the tools are capable of capturing and checking. Bugs due to ambiguous specifications of interconnect logic, for instance, are examples of bugs that are difficult to create security properties for.

Our results, shown in Figure 3, indicate that privilege escalation and denial-of-service (DoS) bugs were the most detected at 60% and 67% respectively. Secret leakage only had a 17% detection rate due to incorrect design specification for one bug, state explosion and the inability to express properties that the tool can assert for the remaining bugs. The code injection bug was undetected by formal techniques. Bugs at the interconnect level of the SoC such as bugs #1 and #2 were especially challenging since they involved a large number of highly complex and inter-connected modules that needed to be loaded and modeled by the tool (see L-1 in Section 3.1). Bug #20, which involves hardware/firmware interactions, was also

detected by neither the state-of-the-art FPV nor SPV since they analyze the RTL in isolation (see L-4 in Section 3.1). We describe these bugs in more detail in Appendix C.

6.3 State-Explosion Problem

Formal verification techniques are quickly driven into state space explosion when analyzing large designs with many states. Many large interconnected RTL modules, like those relevant to bugs #1 and #2, can have states in the order of magnitude of 10^{20}. Even smaller ones, like these used for bugs #3 and #4, can have a very large number of states, as shown in Table 1. When combined, the entire SoC will have a total number of states significantly higher than any of the results in Table 1. Attempting to model the entire SoC drove the tool into state explosion, and it ran out of memory and crashed. Formal verification tools, including those specific to security verification are currently incapable of handling so many states, even when computational resources are increased. This is further aggravated for industry-standard complex SoCs.

Because the entire SoC cannot be modeled and analyzed at once, detecting cross-modular bugs becomes very challenging. Engineers work around this (not fundamentally solve it) by adopting a divide-and-conquer approach and selecting which modules are relevant for the properties being tested and which can be black-boxed or abstracted. However, this is time-consuming, non-automated, error-prone, and requires expertise and knowledge of both the tools and design. By relying on the human factor, the tool can no longer guarantee the absence of bugs for the entire design, which is the original advantage of formal verification.

7 Discussion and Future Work

We now describe why microcode patching is insufficient for RTL bugs while emphasizing the need for advancing the hardware security verification process. We discuss the additional challenges of the overall process, besides the limitations of the industry-standard tools, which is the focus of this work.

7.1 Microcode Patching

While existing industry-grade SoCs support hotfixes by *microcode patching* for instance, this approach is limited to a handful of changes to the instruction set architecture, e.g., modifying the interface of individual complex instructions and adding or removing instructions [25]. Some vulnerabilities cannot even be patched by microcode, such as the recent Spoiler attack [33]. Fundamentally mitigating this requires fixing the hardware of the memory subsystem at the hardware design phase. For legacy systems, the application developer is advised to follow best practices for developing side channel-resilient software[5]. For vulnerabilities that can be patched, patches at this higher abstraction level in the firmware only act as a "symptomatic" fix that circumvent the RTL bug. However, they do not fundamentally patch the bug in the RTL, which is already realized as hardwired logic. Thus, microcode patching is a fallback for RTL bugs discovered after production, when you can not patch the RTL. They may also incur performance impact [6] that could be avoided if the underlying problem is discovered and fixed during design.

7.2 Additional Challenges in Practice

Functional vs. Security Specifications. As described in Section 2, pre- and post-silicon validation efforts are conducted to verify that the implementation fully matches both its functional and security specifications. The process becomes increasingly difficult (almost impossible) as the system complexity increases and specification ambiguity arises. Deviations from specification occur due to either functional or security bugs, and it is important to distinguish between them. While functional bugs generate functionally incorrect results, security bugs are not reflected in functionality. They arise due to unconsidered and corner threat cases that are unlikely to get triggered, thus making them more challenging to detect and cover. It is, therefore, important to distinguish between functional and security specifications, since these are often the references for different verification teams working concurrently on the same RTL implementation.

Specification Ambiguity. Another challenge entails anticipating and identifying all the security properties that are required in a real-world scenario. We analyzed the efficacy of industry-standard tools in a controlled setting—where we have prior knowledge of the bugs. However, in practice hardware validation teams do not have prior knowledge of the bugs. Security specifications are often incomplete and ambiguous, only outlining the required security properties under an assumed adversary model. These specifications are invalidated once the adversary model is changed. This is often the case with IP reuse, where the RTL code for one product is re-purposed for another with a different set of security requirements and usage scenarios. Parameters may be declared multiple times and get misinterpreted by the tools, thus causing bugs to slip undetected. Furthermore, specs usually do not specify bugs and information flows that should not exist, and there is no automated approach to determine whether one is proving the intended properties. Thus, a combination of incomplete or incorrect design decisions and implementation errors can easily introduce bugs to the design.

[5] https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00238.html

[6] https://access.redhat.com/articles/3307751

7.3 Future Research Directions

Through our work, we shed light on the limitations of state-of-the-art verification techniques. In doing so, we hope to motivate further research in advancing these techniques to adequately capture and detect these vulnerabilities.

Although manual RTL inspection is generally useful and can potentially cover a wide array of bugs, its efficacy depends exclusively on the expertise of the engineer. This can be inefficient, unreliable and ad hoc in light of rapidly evolving chip designs. Exhaustive testing of specifications through simulation requires amounts of resources exponential in the size of the input (i.e., design state space) while coverage must be intelligently maximized. Hence, current approaches face severe scalability challenges, as diagnosing software-exploitable bugs that reside deep in the design pipeline can require simulation of trillions of cycles [14]. Our results indicate that it is important to first identify high-risk components due to software exposure, such as password checkers, crypto cores, and control registers, and prioritize analyzing them. Scalability due to complex inter-dependencies among modules is one challenge for detection. Vulnerabilities associated with non-register states (such as caches) or clock-cycle dependencies (i.e., timing flows) are another open problem. Initial research is underway [71] to analyze a limited amount of low-level firmware running on top of a simulated RTL design for information and timing flow violations. However, these approaches are still in their infancy and yet to scale for real-world SoC designs.

8 Related Work

We now present related work in hardware security verification while identifying limitations with respect to detecting HardFails. We also provide an overview of recent software attacks exploiting underlying hardware vulnerabilities.

8.1 Current Detection Approaches

Security-aware design of hardware has gained significance only recently as the critical security threat posed by hardware vulnerabilities became acutely established. Confidentiality and integrity are the commonly investigated properties [19] in hardware security. They are usually expressed using information flow properties between entities at different security levels. Besides manual inspection and simulation-based techniques, systematic approaches proposed for verifying hardware security properties include formal verification methods such as proof assistance, model-checking, symbolic execution, and information flow tracking. We exclude the related work in testing mechanisms, e.g., JTAG/scan-chain/built-in self-test, because they are leveraged for hardware testing **after** fabrication. However, the focus of this work is on verifying the security of the hardware **before** fabrication. Interestingly, this includes verifying that the test mechanisms are correctly implemented in the RTL, otherwise they may constitute security vulnerabilities when used after fabrication (see bugs#9,#10,#11,#12,#16, #26 of the JTAG/debug interface).

Proof assistant and theorem-proving methods rely on mathematically modeling the system and the required security properties into logical theorems and formally proving if the model complies with the properties. VeriCoq [7] based on the Coq proof assistant transforms the Verilog code that describes the hardware design into proof-carrying code.VeriCoq supports the automated conversion of only a subset of Verilog code into Coq. However, this assumes accurate labeling of the initial sensitivity labels of each and every signal in order to effectively track the flow of information. This is cumbersome, error-prone, generates many faluse positives, and does not scale well in practice beyond toy examples. Moreover, timing (and other) side-channel information flows are not modeled. Finally, computational scalability to verifying real-world complex SoCs remains an issue given that the proof verification for a single AES core requires $\approx$ 30 minutes to complete [6].

Model checking-based approaches check a given property against the modeled state space and possible state transitions using provided invariants and predefined conditions. They face scalability issues as computation time scales exponentially with the model and state space size. This can be alleviated by using abstraction to simplify the model or constraining the state space to a bounded number of states using assumptions and conditions. However, this introduces false positives, may miss vulnerabilities, and requires expert knowledge. Most industry-leading tools, such as the one we use in this work, rely on model checking algorithms such as boolean satisfiability problem solvers and property specification schemes, e.g., assertion-based verification to verify the required properties of a given hardware design.

Side-channel leakage modeling and detection remain an open problem. Recent work [76] uses the Murφ model checker to verify different hardware cache architectures for side-channel leakage against different adversary models. A formal verification methodology for SGX and Sanctum enclaves under a limited adversary was introduced in [67]. However, such approaches are not directly applicable to hardware implementation. They also rely exclusively on formal verification and remain inherently limited by the underlying algorithms in terms of scalability and state space explosion, besides demanding particular expertise to use.

Information flow analysis (such as SPV) works by assigning a security label (or a *taint*) to a data input and monitoring the taint propagation. In this way, the designer can verify whether the system adheres to the required security policies. Recently, information flow tracking (IFT) has been shown effective in identifying security vulnerabilities, including timing side channels and information-leaking hardware Trojans.

IFT techniques are proposed at different levels of abstraction: gate-, RT, and language-levels. Gate-level information

flow tracking (GLIFT) [2, 58, 70] performs the IFT analysis directly at gate-level by generating GLIFT analysis logic that is derived from the original logic and operates in parallel to it. Although gate-level IFT logic is easy to automatically generate, it does not scale well. Furthermore, when IFT uses strict non-interference, it taints any information flow conservatively as a vulnerability [34] which scales well for more complex hardware, but generates too many false positives.

At the language level, Caisson [42] and Sapper [41] are security-aware HDLs that use a typing system where the designer assigns security "labels" to each variable (wire or register) based on the security policies required. However, they both require redesigning the RTL using a new hardware description language which is not practical. SecVerilog [22, 75] overcomes this by extending the Verilog language with a dynamic security type system. Designers assign a security label to each variable (wire or register) in the RTL to enable a compile-time check of hardware information flow. However, this involves complex analysis during simulation to reason about the run-time behavior of the hardware state and dependencies across data types for precise flow tracking.

Hardware/firmware co-verification to capture and verify hardware/firmware interactions remains an open challenge and is not available in widely used industry-standard tools. A co-verification methodology [28] addresses the semantic gap between hardware and firmware by modeling hardware and firmware using instruction-level abstraction to leverage software verification techniques. However, this requires modeling the hardware that interacts with firmware into an abstraction which is semi-automatic, cumbersome, and lossy.

While research is underway [71] to analyze a limited amount of low-level firmware running on top of a simulated RTL design these approaches are still under development and not scalable. Current verification approaches focus on register-state information-flow analysis, e.g., to monitor whether sensitive locations are accessible from unprivileged signal sources. Further research is required to explicitly model non-register states and timing explicitly alongside the existing capabilities of these tools.

8.2 Recent Attacks

We present and cautiously classify the underlying hardware vulnerabilities of recent cross-layer exploits (see Table 2 in Appendix B), using the categories introduced in 3.1. We do not have access to proprietary processor implementations, so our classification is only based on our deductions from the published technical descriptions. Yarom et al. demonstrate that software-visible side channels can exist even below cache-line granularity in CacheBleed [74]–undermining a core assumption of prior defenses, such as scatter-gather [9]. MemJam [45] exploits false read-after-write dependencies in the CPU to maliciously slow down victim accesses to memory blocks within a cache line. We categorize the underlying vulnerabilities of CacheBleed and MemJam as potentially hard to detect in RTL due to the many cross-module connections involved and the timing-flow leakage. The timing flow leakage is caused by the software triggering clock cycle differences in accesses that map to the same bank below cache line granularity, thus breaking constant-time implementations.

The TLBleed [23] attack shows how current TLB implementations can be exploited to break state-of-the-art cache side-channel protections. As described in Section 4, TLBs are typically highly interconnected with complex processor modules, such as the cache controller and memory management unit, making vulnerabilities therein very hard to detect through automated verification or manual inspection.

BranchScope [20] extracts information through the directional branch predictor, thus bypassing software mitigations that prevent leakage via the BTB. We classify it as a cache-state gap in branch prediction units, which is significantly challenging to detect using existing RTL security verification tools, which cannot capture and verify cache states. Meltdown [43] exploits speculative execution on modern processors to completely bypass all memory access restrictions. Van Bulck et al. [72] also demonstrated how to apply this to Intel SGX. Similarly, Spectre [37] exploits out-of-order execution across different user-space processes as arbitrary instruction executions would continue during speculation. We recognize these vulnerabilities are hard to detect due to scalability challenges in existing tools, since the out-of-order scheduling module is connected to many subsystems in the CPU. Additionally, manually inspecting these interconnected complex RTL modules is very challenging and cumbersome.

CLKScrew [69] abuses low-level power-management functionality that is exposed to software to induce faults and glitches dynamically at runtime in the processor. We categorize CLKScrew to have vulnerable hardware-firmware interactions and timing-flow leakage, since it directly exposes clock-tuning functionality to attacker-controlled software.

9 Conclusion

Software security bugs and their impact have been known for many decades, with a spectrum of established techniques to detect and mitigate them. However, the threat of hardware security bugs has only recently become significant as cross-layer exploits have shown that they can completely undermine software security protections. While some hardware bugs can be patched with microcode updates, many cannot, often leaving millions of affected chips in the wild. In this paper, we presented the first testbed of RTL bugs and systematically analyzed the effectiveness of state-of-the-art formal verification techniques, manual inspection and simulation methods in detecting these bugs. We organized an international hardware security competition and an in-house study. Our results have shown that 54 teams were only able to detect 61% of the total number of bugs, while with industry-leading formal verification techniques, we were only able to detect 48% of

the bugs. We showcase that the grave security impact of many of these undetected bugs is only further exacerbated by being software-exploitable.

Our investigation revealed the limitations of state-of-the-art verification/detection techniques with respect to detecting certain classes of hardware security bugs that exhibit particular properties. These approaches remain limited in the face of detecting vulnerabilities that require capturing and verifying complex cross-module inter-dependencies, timing flows, cache states, and hardware-firmware interactions. While these effects are common in SoC designs, they are difficult to model, capture, and verify using current approaches. Our investigative work highlights the necessity of treating the detection of hardware bugs as significantly as that of software bugs. Through our work, we highlight the pressing call for further research to advance the state of the art in hardware security verification. Particularly, our results indicate the need for increased scalability, efficacy and automation of these tools, making them easily applicable to large-scale commercial SoC designs—without which software protections are futile.

Acknowledgments

We thank our anonymous reviewers and shepherd, Stephen Checkoway, for their valuable feedback. The work was supported by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), the German Research Foundation (DFG) by CRC 1119 CROSSING P3, and the Office of Naval Research (ONR Award #N00014-18-1-2058). We would also like to acknowledge the co-organizers of Hack@DAC: Dan Holcomb (UMass-Amherst), Siddharth Garg (NYU), and Sourav Sudhir (TAMU), and the sponsors of Hack@DAC: the National Science Foundation (NSF CNS-1749175), NYU CCS, Mentor - a Siemens Business and CROSSING, as well as the participants of Hack@DAC.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. *ACM conference on Computer and communications security*, pages 340–353, 2005.

[2] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design. *Design, Automation & Test in Europe*, pages 1695–1700, 2017.

[3] ARM. Security technology building a secure system using trustzone technology (white paper). http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[4] R. Armstrong, R. Punnoose, M. Wong, and J. Mayo. Survey of Existing Tools for Formal Verification. Sandia National Laboratories https://prod.sandia.gov/techlib-noauth/access-control.cgi/2014/1420533.pdf, 2014.

[5] Averant. Solidify. http://www.averant.com/storage/documents/Solidify.pdf, 2018.

[6] M.-M. Bidmeshki, X. Guo, R. G. Dutta, Y. Jin, and Y. Makris. Data Secrecy Protection Through Information Flow Tracking in Proof-Carrying Hardware IP—Part II: Framework Automation. *IEEE Transactions on Information Forensics and Security*, 12(10):2430–2443, 2017.

[7] M.-M. Bidmeshki and Y. Makris. VeriCoq: A Verilog-to-Coq Converter for Proof-Carrying Hardware Automation. *IEEE International Symposium on Circuits and Systems*, pages 29–32, 2015.

[8] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. *Network and Distributed System Security Symposium (NDSS)*, 2019.

[9] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, 2006.

[10] Cadence. Incisive Enterprise Simulator. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html, 2014.

[11] Cadence. JasperGold Formal Verification Platform. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html, 2014.

[12] Cadence. JasperGold Security Path Verification App. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html, 2018. Last accessed on 09/09/18.

[13] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. *USENIX Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[14] D. P. Christopher Celio, Krste Asanovic. The Berkeley Out-of-Order Machine. https://riscv.org/wp-content/uploads/2016/01/Wed1345-RISCV-Workshop-3-BOOM.pdf, 2016.

[15] Cisco. Cisco: Strengthening Cisco Products. https://www.cisco.com/c/en/us/about/security-center/security-programs/secure-development-lifecycle.html, 2017.

[16] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. *Tools for Practical Software Verification*, 2012.

[17] K. Conger. Apple announces long-awaited bug bounty program. https://techcrunch.com/2016/08/04/apple-announces-long-awaited-bug-bounty-program/, 2016.

[18] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. *USENIX Security Symposium*, pages 857–874, 2016.

[19] O. Demir, W. Xiong, F. Zaghloul, and J. Szefer. Survey of approaches for security verification of hardware/software systems. https://eprint.iacr.org/2016/846.pdf, 2016.

[20] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, D. 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.

[21] F. Farahmandi, Y. Huang, and P. Mishra. Formal Approaches to Hardware Trust Verification. *The Hardware Trojan War*, 2018.

[22] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 555–568, 2017.

[23] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. *USENIX Security Symposium*, 2018.

[24] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379, 2016.

[25] M. Hicks, C. Sturton, S. T. King, and J. M. Smith. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS. ACM, 2015.

[26] M. Howard and S. Lipner. The Security Development Lifecycle. *Microsoft Press Redmond*, 2006.

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

[28] B.-Y. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik. Formal Security Verification of Concurrent Firmware in SoCs Using Instruction-level Abstraction for Hardware. *ACM Annual Design Automation Conference*, pages 91:1–91:6, 2018.

[29] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. *Symposium on Security and Privacy*, 2013.

[30] F. Inc. Common Vulnerability Scoring System v3.0. https://www.first.org/cvss/cvss-v30-specification-v1.8.pdf, 2018.

[31] Intel. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx, 2016. Last accessed on 09/05/18.

[32] Intel. Intel Bug Bounty Program. https://www.intel.com/content/www/us/en/security-center/bug-bounty-program.html, 2018.

[33] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. https://arxiv.org/abs/1903.00446, 2019.

[34] R. Kastner, W. Hu, and A. Althoff. Quantifying Hardware Security Using Joint Information Flow Analysis. *IEEE Design, Automation & Test in Europe*, pages 1523–1528, 2016.

[35] H. Khattri, N. K. V. Mangipudi, and S. Mandujano. Hsdl: A security development lifecycle for hardware technologies. *IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 116–121, 2012.

[36] 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.

[37] 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. http://arxiv.org/abs/1801.01203, 2018.

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

[39] D. Lee. Keystone enclave: An open-source secure enclave for risc-v. https://keystone-enclave.org/, 2018.

[40] Lenovo. Lenovo: Taking Action on Product Security. https://www.lenovo.com/us/en/product-security/about-lenovo-product-security, 2017.

[41] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: A Language for Hardware-level Security Policy Enforcement. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–112, 2014.

[42] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 46(6):109–120, 2011.

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. https://arxiv.org/abs/1801.01207, 2018.

[44] Mentor. Questa Verification Solution. https://www.mentor.com/products/fv/questa-verification-platform, 2018.

[45] A. Moghimi, T. Eisenbarth, and B. 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.

[46] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[47] NIST. HP: Remote update feature in HP LaserJet printers does not require password. https://nvd.nist.gov/vuln/detail/CVE-2004-2439, 2004.

[48] NIST. Microsoft: Hypervisor in Xbox 360 kernel allows attackers with physical access to force execution of the hypervisor syscall with a certain register set, which bypasses intended code protection. https://nvd.nist.gov/vuln/detail/CVE-2007-1221, 2007.

[49] NIST. Apple: Multiple heap-based buffer overflows in the AudioCodecs library in the iPhone allows remote attackers to execute arbitrary code or cause DoS via a crafted AAC/MP3 file. https://nvd.nist.gov/vuln/detail/CVE-2009-2206, 2009.

[50] NIST. Broadcom Wi-Fi chips denial of service. https://nvd.nist.gov/vuln/detail/CVE-2012-2619, 2012.

[51] NIST. Vulnerabilities in Dell BIOS allows local users to bypass intended BIOS signing requirements and install arbitrary BIOS images. https://nvd.nist.gov/vuln/detail/CVE-2013-3582, 2013.

[52] NIST. Google: Escalation of Privilege Vulnerability in MediaTek WiFi driver. https://nvd.nist.gov/vuln/detail/CVE-2016-2453, 2016.

[53] NIST. Samsung: Page table walks conducted by MMU during Virtual to Physical address translation leaves in trace in LLC. https://nvd.nist.gov/vuln/detail/CVE-2017-5927, 2017.

[54] NIST. AMD: Backdoors in security co-processor ASIC. https://nvd.nist.gov/vuln/detail/CVE-2018-8935, 2018.

[55] NIST. AMD: EPYC server processors have insufficient access control for protected memory regions. https://nvd.nist.gov/vuln/detail/CVE-2018-8934, 2018.

[56] NIST. Buffer overflow in bootrom recovery mode of nvidia tegra mobile processors. https://nvd.nist.gov/vuln/detail/CVE-2018-6242, 2018.

[57] J. Oberg. Secure Development Lifecycle for Hardware Becomes an Imperative. https://www.eetimes.com/author.asp?section_id=36&doc_id=1332962, 2018.

[58] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. *IEEE/ACM Design Automation Conference*, pages 244–247, 2010.

[59] PULP Platform. Ariane. https://github.com/pulp-platform/ariane, 2018.

[60] PULP Platform. Pulpino. https://github.com/pulp-platform/pulpino, 2018.

[61] PULP Platform. Pulpissimo. https://github.com/pulp-platform/pulpissimo, 2018.

[62] Qualcomm. Qualcomm Announces Launch of Bounty Program. https://www.qualcomm.com/news/releases/2016/11/17/qualcomm-announces-launch-bounty-program-offering-15000-usd-discovery, 2018.

[63] Samsung. Rewards Program. https://security.samsungmobile.com/rewardsProgram.smsb, 2018.

[64] M. Seaborn and T. Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat*, 15, 2015.

[65] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *ACM Symposium on Computer and Communication Security*, pages 552–561, 2007.

[66] O. Solutions. OneSpin 360. https://www.onespin.com/fileadmin/user_upload/pdf/datasheet_dv_web.pdf, 2013.

[67] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia. A Formal Foundation for Secure Remote Execution of Enclaves. *ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450, 2017.

[68] Sunny .L He and Natalie H. Roe and Evan C. L. Wood and Noel Nachtigal and Jovana Helms. Model of the Product Development Lifecycle. https://prod.sandia.gov/techlib-noauth/access-control.cgi/2015/159022.pdf, 2015.

[69] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy managemen. *USENIX Security Symposium*, pages 1057–1074, 2017.

[70] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete Information Flow Tracking from the Gates Up. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 109–120, 2009.

[71] Tortuga Logic. Verifying Security at the Hardware/Software Boundary. http://www.tortugalogic.com/unison-whitepaper/, 2017.

[72] J. Van Bulck, F. Piessens, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[73] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, 2014.

[74] Y. Yarom, D. Genkin, and N. Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017. 10.1007/s13389-017-0152-y.

[75] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, 2015.

[76] T. Zhang and R. B. Lee. New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels. *ACSAC*, pages 96–105, 2014.

Appendix

A Ariane Core and RTL Hierarchy

Figure 4 shows the high-level microarchitecture of the Ariane core to visualize its complexity. This RISC-V core is far less complex than an x86 or ARM processor and their more sophisticated microarchitectural and optimization features.

Figure 5 illustrates the hierarchy of the RTL components of the Ariane core. This focuses only on the core and excludes all uncore components, such as the AXI interconnect, peripherals, the debug module, boot ROM, and RAM.

B Recent Microarchitectural Attacks

We reviewed recent microarchitectural attacks with respect to existing hardware verification approaches and their limitations. We observe that the underlying vulnerabilities would be difficult to detect due to the properties that they exhibit, rendering them as potential HardFails. We do not have access to their proprietary RTL implementation and cannot inspect the underlying vulnerabilities. Thus, we only infer from the published technical descriptions and errata of these attacks the nature of the underlying RTL issues. We classify in Table 2 the properties of these vulnerabilities that represent challenges for state-of-the-art hardware security verification.

C Details on the Pulpissimo Bugs

We present next more detail on some of the RTL bugs used in our investigation.

Bugs in crypto units and incorrect usage: We extended the SoC with a faulty cryptographic unit with a multiplexer to select between AES, SHA1, MD5, and a temperature sensor. The multiplexer was modified such that a race condition occurs if more than one bit in the status register is enabled, causing unreliable behavior in these security critical modules.

Furthermore, both SHA-1 and MD5 are outdated and broken cryptographic hash functions. Such bugs are not detectable by formal verification, since they occur due to a specification/design issue and not an implementation flaw, therefore they are out of the scope of automated approaches and formal verification methods. The cryptographic key is

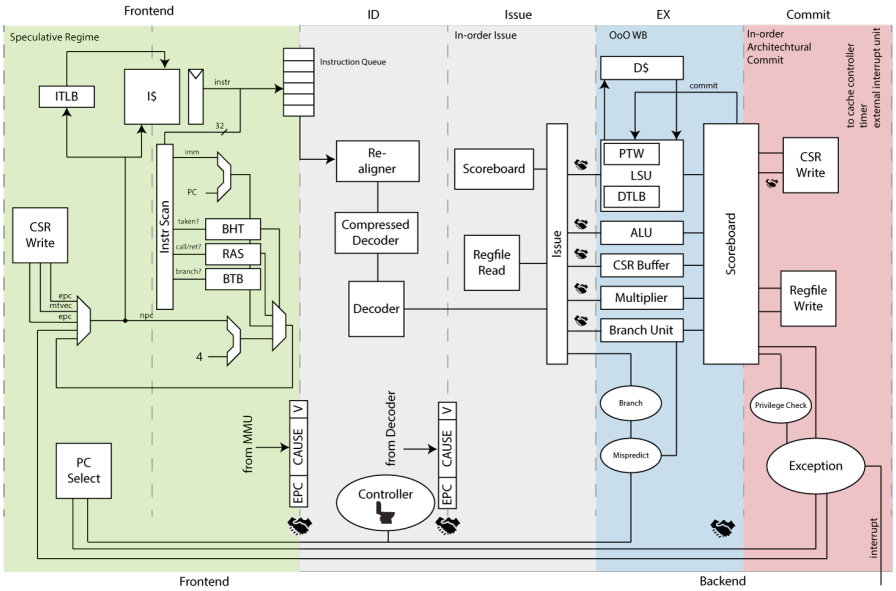


FIGURE 4: High-level architecture of the Ariane core [59].

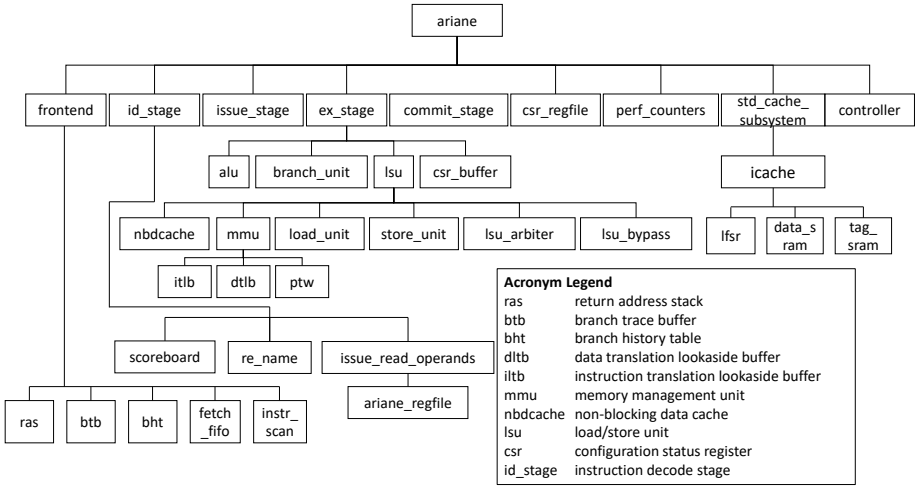


FIGURE 5: Illustration of the RTL module hierarchy of the Ariane core.

Attack	Privilege Level	Memory Corruption	Information Leakage	Cross-modular	HW/FW-Interaction	Cache-State Gap	Timing-Flow Gap	**HardFail**
Cachebleed [74]	unprivileged	✗	✓	✗	✗	✗	✓	✓
TLBleed [23]	unprivileged	✗	✓	✓	✗	✓	✓	✓
BranchScope [20]	unprivileged	✗	✓	✗	✗	✓	✗	✓
Spectre [37]	unprivileged	✗	✓	✓	✗	✓	✗	✓
Meltdown [43]	unprivileged	✗	✓	✓	✗	✓	✗	✓
MemJam [45]	supervisor	✗	✓	✓	✗	✗	✓	✓
CLKScrew [69]	supervisor	✓	✓	✗	✓	✗	✓	✓
Foreshadow [72]	supervisor	✓	✓	✓	✓	✓	✗	✓

TABLE 2: Classification of the underlying vulnerabilities of recent microarchitectural attacks by their HardFail properties.

stored and read from unprotected memory, allowing an attacker access to the key. The temperature sensor register value is incorrectly muxed as output instead of the crypto engine output and vice versa, which are illegal information flows that could compromise the cryptographic operations.

LISTING 1: **Incorrect use of crypto RTL:** The key input for the AES (g_input) is connected to signal b. This signal is then passed through various modules until it connects directly to a tightly coupled memory in the processor.

```
input logic [127:0] b,
...
aes_1cc aes(
  .clk(0),
  .rst(1),
  .g_input(b),
  .e_input(a),
  .o(aes_out)
  );
```

Bugs in security modes: We replaced the standard PULP_SECURE parameter in the riscv_cs_registers and riscv_int_controller modules with another constant parameter to permanently disable the security/privilege checks for these two modules. Another bug we inserted is switching the write and read protections for the AXI bus interface, causing erroneous checks for read and write accesses.

Bugs in the JTAG module: We implemented a JTAG password-checker and injected multiple bugs in it, including the password being hardcoded in the password checking file. The password checker also only checks the first 31 bits, which reduces the computational complexity of brute-forcing the password. The password checker does not reset the state of the correctness of the password when an incorrect bit is detected, allowing for repeated partial checks of passwords to end up unlocking the password checker. This is also facilitated by the fact that the index overflows after the user hits bit 31, allowing for an infinite cycling of bit checks.

D Exploiting Hardware Bugs From Software

We now explain how one of our hardware bugs can be exploited in real-world by software. This RTL vulnerability manifests in the following way. When an error signal is generated on the memory bus while the underlining logic is still handling an outstanding transaction, the next signal to be handled will instead be considered operational by the module unconditionally. This lets erroneous memory accesses slip through hardware checks at runtime. Armed with the knowledge about this vulnerability, an adversary can force memory access errors to evade the checks. As shown in Figure 6, the memory bus decoder unit (unit of the memory interconnect) is assumed to have the bug. This causes errors to be ignored

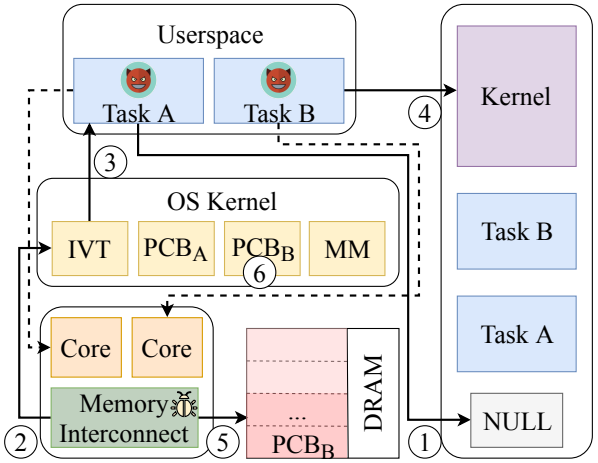


FIGURE 6: Our attack exploits a bug in the implementation of the memory bus of the PULPissimo SoC: by ① *spamming* the bus with invalid transactions an adversary can make ④ malicious write requests be set to `operational`.

under certain conditions (see bug number #7 in Table 1). In the first step ①, the attacker generates a user program (Task A) that registers a dummy signal handler for the segmentation fault (`SIGSEGV`) access violation. Task A then executes a loop with ② a faulting memory access to an invalid memory address (e.g., `LW x5, 0x0`). This will generate an error in the memory subsystem of the processor and issue an invalid memory access interrupt (i.e., `0x0000008C`) to the processor. The processor raises this interrupt to the running software (in this case the OS), using the pre-configured interrupt handler routines in software. The interrupt handler in the OS will then forward this as a signal to the faulting task ③, which keeps looping and continuously generating invalid accesses. Meanwhile, the attacker launches a separate Task B, which will then issue a single memory access ④ to a privileged memory location (e.g., `LW x6, 0xf77c3000`). In this situation, multiple outstanding memory transactions will be generated on the memory bus, all of which but one will be flagged as faulty by the address decoder. An invalid memory access will always proceed the single access of Task B. Due to the bug in the memory bus address decoder, ⑤ the malicious memory access will become `operational` instead of triggering an error. Thus, the attacker can issue read and write instructions to arbitrary privileged (and unprivileged) memory by forcing the malicious illegal access to be preceded with a faulty access. Using this technique the attacker can eventually leverage this read-write primitive, e.g., ⑥ to escalate privileges by writing the process control block (PCB_B) for his task to elevate the corresponding process to root. This bug leaves the attacker with access to a root process, gaining control over the entire platform and potentially compromising all the processes running on the system.