Enclave Computing Paradigm: Hardware-assisted Security Architectures & Applications

Brasser, Franz Ferdinand Peter

(2020)

Enclave Computing Paradigm: Hardware-assisted Security Architectures & Applications

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

MSc. Franz Ferdinand Peter Brasser
Geboren am 27. April 1984 in Wolfenbüttel, Deutschland

Referenten:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (Erstreferent)
Prof. Gene Tsudik, PhD (Zweitreferent)

Tag der Einreichung: 06. April 2020
Tag der Disputation: 23. Juni 2020

TECHNISCHE
UNIVERSITÄT
DARMSTADT

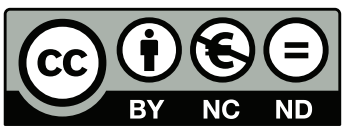
System Security Lab
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

Franz Ferdinand Peter Brasser:
Enclave Computing Paradigm: Hardware-assisted Security Architectures & Applications

PHD REFEREES:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (1st PhD Referee)
Prof. Gene Tsudik, PhD (2nd PhD Referee)

Darmstadt, Germany April 2020



ABSTRACT

Hardware-assisted security solutions, and the isolation guarantees they provide, constitute the basis for the protection of modern software systems. Hardware-enforced isolation of individual components reduces complexity of the overall software as well as the size and complexity of the individual components. The basic idea is that a reduction in complexity minimizes the probability of vulnerabilities in the software, thus strengthening the system's security.

In classical system architectures, an application's security depends on the security of all privileged system entities, for example the Operating System. The Trusted Execution Environment (TEE) concept overcomes the dependence of security critical components on the systems overall security. TEEs provide isolated compartments within a single system, allowing isolated operation of a system's individual components and applications.

The *enclave computing paradigm* enhances the TEE concept by enabling self-contained isolation of system components and applications, fulfilling the needs of modern software. It enables novel use cases by providing many parallel mutually isolated TEE-instances without the need to rely on complex privileged entities.

The TEE solutions developed by industry and deployed in today's systems follow distinct design approaches and come with various limitations. ARM TrustZone, which is widely available in mobile devices, is fundamentally limited to a single isolation domain. Intel's TEE solution Software Guard Extensions (SGX) provides multiple mutually isolated execution environments, called *enclaves*. However, SGX enclaves face severe threats, in particular side-channel leakage, that can void its security guarantees. Preventing side-channel leakage from enclaves in a universal and efficient way is a non-trivial problem. Nevertheless, these deployed TEE solutions enable various novel applications. However, different TEE architectures come with diverse properties and features that require special consideration in the design of TEE applications.

Security architectures for embedded systems face additional challenges that have not been solved, neither by industry nor by academic research. These security architectures need to be compliant with and need to preserve all functional requirements of an embedded system. Since network-connected embedded devices are increasingly used in safety critical systems, such as industrial control systems or automotive scenarios, security architectures that combine safety and security aspects are vitally needed.

Remote Attestation (RA) is a security service that relies on the isolation guarantees of TEEs. It is of particularly high relevance for connected embedded systems. It allows trust establishment between these devices enabling their reliable collaboration in large connected systems. However, many aspects of RA, such as its scalability in large networks or its applicability in autonomous connected systems, are unexplored.

In this dissertation, we present novel isolation architectures that bring the enclave computing paradigm to mobile and embedded platforms. We present the first security

architecture for small embedded systems that provides isolated execution enclaves and real-time guarantees. Moreover, we present a novel multi-TEE security architecture for TrustZone-systems bringing the enclave computing paradigm to mobile systems, overcoming TrustZone's fundamental limitation.

Furthermore, we deal with Intel SGX's vulnerability to side-channel attacks. We demonstrate the severity of side-channel leakage due to observable memory access patterns of SGX enclaves. To counter side-channel attacks, we present solutions that hide memory access patterns of enclaves for both accesses to enclave-external memory as well as access patterns within enclaves' private memory.

We present two TEE-applications that follow different design approaches, leveraging the specific capabilities of Intel SGX and ARM TrustZone, respectively. We introduce a cloud-based machine learning solution that enables privacy-preserving speech recognition utilizing isolated execution enclaves. We also demonstrate the limitations of the enclave computing paradigm and show a (remote) policy enforcement solution for mobile devices, which requires an isolated execution environment with elevated privileges.

Additionally, we investigate novel RA schemes, which tackle many important aspects of RA that are highly relevant in emerging connected systems. We develop solutions to prevent the misuse of remote attestation for Denial-of-Service (DoS) attacks and present the first efficient multi-prover attestation scheme. Furthermore, we introduce the concept of data integrity attestation, which allows the efficient and reliable collaboration of autonomous connected devices.

ZUSAMMENFASSUNG

Hardware-basierte Sicherheitslösungen und die durch sie gebotenen Sicherheitsgarantien bilden die Basis zum Schutz moderner Softwaresysteme. Durch die Isolation von Systemkomponenten durch Hardwaremechanismen können die Größe und Komplexität des Gesamtsystems als auch der individuellen Komponenten reduziert werden. Somit sorgt Isolation für eine Verbesserung der Systemsicherheit, denn eine Reduktion der Komplexität führt in der Regel gleichzeitig zu einer Reduktion der Wahrscheinlichkeiten von Schwachstellen in der Software.

In herkömmlichen Computerarchitekturen hängt die Sicherheit einer Anwendung von der Sicherheit aller privilegierten Systemkomponenten, wie etwa dem Betriebssystem, ab. Das Konzept von Trusted Execution Environments (TEEs) überwindet diese Abhängigkeit, d.h. sicherheitskritische Systemkomponenten sind nicht mehr von der Sicherheit des Gesamtsystems abhängig. TEEs stellen isolierte Bereiche innerhalb eines einzelnen Systems bereit, diese erlauben die abgeschottete Ausführung der individuellen Komponenten eines Systems.

Das *Enclave Computing Paradigma* entwickelt das TEE Konzept weiter und bietet in sich geschlossene Isolationsbereiche für Systemkomponenten und Anwendungen. Somit erfüllt es die Anforderungen moderner Software. Es ermöglicht neuartige Anwendungsfälle, da viele parallele, wechselseitig isolierte TEE-Instanzen geschaffen werden, ohne auf komplexe privilegierte Komponenten angewiesen zu sein.

Die durch die Industrie entwickelten und aktuell verfügbaren TEE-Lösungen folgen unterschiedlichen Ansätzen und leiden unter verschiedenen Nachteilen. ARM TrustZone, das in vielen mobilen Systemen integriert ist, hat die zentrale Einschränkun, dass es nur eine einzige Isolationsdomäne bietet. Intel Software Guard Extensions (SGX) bietet hingegen viele, wechselseitig isolierte Ausführungsumgebungen, die *Encalves* genannt werden. Allerdings werden SGX-Enclaves von schwerwiegenden Schwachstellen bedroht, die ihre Sicherheitsgarantien zunichtemachen können. Vor allem Angriffe, die Seitenkanäle ausnutzen, bedrohen SGX. Eine generelle und effiziente Lösung zu finden, die Seitenkanalangriffe auf Enclaves verhindern kann, stellt dabei ein nicht-triviales Problem dar. Trotz ihrer Probleme ermöglichen diese bereits verfügbaren TEE-Lösungen viele neue Anwendungen. Bei der Konzeptionierung und Entwicklung von TEE-Anwendungen müssen allerdings die spezifischen Eigenschaften der jeweiligen TEE-Architekturen berücksichtigt werden.

Bei Sicherheitsarchitekturen für eingebettete Systeme ergeben sich zusätzliche Herausforderungen, die überwunden werden müssen. Dies ist bislang weder der Industrie noch der akademischen Forschung gelungen, da diese Sicherheitsarchitekturen die funktionalen Eigenschaften des zugrundeliegenden Systems erhalten müssen. Vernetzte eingebettete System werden zunehmend in sicherheitskritischen Bereichen, etwa in der Industrieautomatisierung oder in der Fahrzeugsteuerung, eingesetzt. Deshalb sind hier

Sicherheitsarchitekturen, die Funktionalität und Sicherheit gleichermaßen gewährleisten, von entscheidender Bedeutung.

Remote Attestation (RA) ist ein Sicherheitsmechanismus, der auf den Isolationsgarantien von TEEs aufbaut. RA ist besonders wichtig im Kontext vernetzter eingebetteter Systeme. Es erlaubt den Aufbau von Vertrauensverhältnissen zwischen Systemen und ermöglicht so die verlässliche Kollaboration dieser in weitvernetzten Anlagen. Allerdings sind viele Aspekte von RA, etwa zur Skalierbarkeit in großflächigen Netzwerken oder zur Anwendbarkeit in autonomen Systemen, noch unerforscht.

In dieser Dissertation stellen wir neue Isolationsarchitekturen vor, die das Enclave-Konzept für Mobilsysteme und eingebettete Systeme umsetzen. Wir präsentieren die erste Sicherheitsarchitektur für kleine eingebettete Systeme, die isolierte Ausführungsumgebungen ermöglicht und gleichzeitig Echtzeitgarantien bietet. Für mobile Systeme präsentieren wir eine neuartige Sicherheitsarchitektur, die viele unabhängige TEE-Instanzen auf TrustZone-basierten Systemen ermöglicht, und somit die zentrale Einschränkung von TrustZone überwindet.

Weiterhin betrachten wir die Verwundbarkeit von Intel SGX durch Seitenkanalangriffe. Wie demonstrieren die Ernsthaftigkeit von Angriffen, welche die Speicherzugriffsmuster einer SGX-Enclave überwachen. Zur Abwehr von Seitenkanalangriffen präsentieren wir Lösungen, die sowohl den Zugriff einer Enclave auf externen Speicher wie auch die Zugriffsmuster im Enclave-eigenen Speicher verbergen.

Wie präsentieren zwei TEE-Anwendungen, die unterschiedlichen Konzepten folgen, dabei nutzen sie die spezifischen Möglichkeiten von Intel SGX bzw. ARM TrustZone. Wir stellen eine Cloud-basierte Lösung für maschinelles Lernen vor, die Privatsphäre-schützende Spracherkennung durch die Nutzung von Enclaves ermöglicht. Zudem zeigen wir die Grenzen des Enclave Computing Paradigmas anhand einer Lösung auf, die ortsabhängige Nutzungsregeln für Mobilgeräte durchsetzen kann, welche jedoch nur mit einer isolierten Ausführungsumgebung mit erhöhten Privilegien umsetzbar ist.

Darüber hinaus entwickeln wir neue RA Mechanismen, die wichtige Aspekte betrachten, die vor allem im Kontext von künftigen vernetzten Systemen hohe Relevanz haben. Wir entwickeln Lösungen, die den Missbrauch von Remote Attestation für Dienstverweigerungsattacken (Denial-of-Service) verhindern, und wir präsentieren das erste Protokoll zur effizienten gleichzeitigen Attestierung vieler Geräte. Weiterhin führen wir das Konzept der Datenintegritätsattestierung ein, welches die verlässliche und effiziente Kollaboration von autonom interagierenden Geräten ermöglicht.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Prof. Ahmad-Reza Sadeghi for his guidance and support during my doctorate. His dedication to research has inspired me already during my master studies and encouraged me to pursue a PhD. He supported my research in various ways, most importantly, he gave me the freedom to follow my own research ideas, he connected me with many well established researchers worldwide, and he made me a part of his System Security Lab for almost a decade – first as a student assistant and later as a research assistant.

I am very honored to have Prof. Gene Tsudik as my thesis co-advisor and would like to thank him for his detailed feedback on this thesis as well as for the inspiring discussions in the course of our collaborations.

During my time at the System Security Lab I had the opportunity and pleasure to have worked together with many professors, colleagues, and students. I had many interesting projects with them and I want to thank everyone for their contribution to the success of these collaborations. My thanks go to the external collaborates from academia and industry as well as to my colleagues and students at TU Darmstadt. I want to express my thanks to the team members of the System Security Lab that were always supportive and contributed to a pleasant work atmosphere.

Finally, I would like to thank my family and friends for their support before and during my doctorate. Special thanks go to my parents. They always supported me and believed in me. Thanks to them I was able to study, even after some throwbacks. Without their encouragement I could not have made it.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

In an increasingly connected world of computing systems that become ever more complex, ensuring the security of these system is a growing challenge. Our experiences from decades of computer security show that building *secure* computing systems is a hard problem, which – despite all efforts – has not been solved. In particular, software vulnerabilities continuously endanger our systems [328, 232, 146, 347, 76, 195, 98, 196], and recently also side-channel-based attacks have gained prominence [225, 247, 390].

Today's devices, due to their ubiquitous connectivity, typically do not operate self-contained. Instead they largely depend on services provided by other entities, for instance cloud services, that are central for many applications and use cases. This dependence implies strong trust requirements, which are hard to establish with today's systems. Connected systems are often built, maintained, and operated by mutually distrusting parties. Furthermore, compromised, i. e., systems controlled by an adversary, can negatively impact a connected overall system through all sorts of malevolent actions.

For instance, cloud services imply extensive trust in the cloud provider that must act honestly and ensure that the cloud infrastructure cannot be compromised by an adversary. The strict hierarchical privilege architecture in classical computer platforms leads to large and complex software systems that need to operate correctly to provide the desired protections, e. g., to protect applications operating on sensitive data from unauthorized accesses. Furthermore, the correct behavior of a system cannot be validated by external parties, for instance, a cloud client cannot verify that the cloud infrastructure is not compromised.

Hardware-assisted security promises to solve many long-existing problems of vulnerable software. Hardware security features are used to store and protect sensitive information such as cryptographic keys, to isolate a minimal set of security critical software, or to implement security critical functions directly in hardware. The assumptions are that hardware is less likely to have vulnerabilities and that it is desirable to minimize the amount and size of security critical software. Minimizing the security critical software reduces its complexity, which will also reduce the likelihood for vulnerabilities in the software. Furthermore, hardware features are commonly used as Root of Trust (RoT) to establish trust relations between different entities. The immutable nature of hardware preserves its integrity and, thus, helps to sustain trust placed in it.

The development, standardization and deployment of hardware security primitives has mainly been propelled by industry. The security and privacy concerns of users and businesses with regard to cloud computing have been the focus of research for many years. Early solutions relied on Trusted Platform Modules (TPMs) and software isolation based on trusted hypervisors [266]. Also, in mobile devices, Trusted Execution Environments (TEEs) are widely deployed, ARM TrustZone [24] is the dominant solution

in this sector. However, its design is intended to divide a platform into only two isolation domains, one domain for insecure legacy applications and a second domain for security critical functions. In order to protect the security domain, access to it is highly restricted. In this work, we present SANCTUARY, our security architecture providing multiple mutually isolated TEEs on TrustZone-based systems [73].

With the introduction of Intel's TEE, called Software Guard Extensions (SGX) [267, 193, 20], many solutions have been published aiming to secure cloud applications using TEEs [41, 339, 33, 355]. SGX provides multiple mutually isolated TEEs, called *enclaves*, that promise code integrity, as well as integrity and confidentiality for their data.

Unfortunately, SGX's isolation guarantees can be circumvented via side-channel attacks [68, 340, 277, 174, 184, 405, 109, 412, 183, 176, 239, 143], as we demonstrate in this work (cf. Section 4.1). To improve the security of industry TEEs, such as SGX, additional measures must be taken. SGX enclaves must be protected against various side-channel attacks. The access patterns to externally managed resources, for example data stored in a database, can reveal confidential information meant to be protected inside an enclave. Similarly, the memory access patterns of an enclave, e. g., observed through an enclave's cache usage, can disclose sensitive information. With HardIDX we present an SGX-protected database index that hides access patterns to enclave-external resources [149, 151]. DR.SGX is our novel data randomization scheme to obfuscate enclaves' cache access patters in order to prevent side-channel leakage [72].

The progression of the Internet of Things (IoT) has let embedded devices become the heart of many security critical applications. Embedded devices control almost all aspects of our lives as they are connecting home appliances, vehicles, medical devices, industrial facilities, critical infrastructure, and many more. Despite some efforts in academia and first steps by industry, TEE solutions for embedded systems are not widely available. The functional requirements for embedded systems as well as their constrained resources pose additional challenges for the development of comprehensive embedded systems security architectures. In particular, real-time capabilities are crucial for many safety critical systems. Our security architecture TYTAN presents the first TEE solution for small embedded devices providing enclave-like execution environments and real-time guarantees, allowing TYTAN to be applied in many safety critical applications [62].

TEEs provide security guarantees for individual services that execute isolated on a single device. However, the ubiquitous connectivity of today's devices demands solutions for the secure collaboration of devices.

Relying on centralized services to secure connected large-scale systems is not desirable. Centralized services have various disadvantages, e. g., they often do not scale well and represent a single point of failure in the overall system. Therefore, it is desirable for future systems to be composed of interconnected devices that act autonomously and collaboratively work together towards a common goal without relying on central entities or services. For instance, in connected vehicle scenarios, where cars must make decisions based on information provided by other traffic participants, internet connectivity, e. g., to a cloud service, cannot always be assumed and vehicles have to directly communicate with each other.

Remote Attestation (RA) is a powerful security service that allows trust establishment between remote parties in connected systems, i. e., a prover device can attest to another device, called verifier, its current state. However, RA has limitations regarding scalability, while safety critical systems have high availability requirements that can conflict with RA protocols and systems. To address this conflict RA has to be considered from the prover's perspective in scenarios where the verifier might not be completely trusted and try to misuse the attestation primitive. In this work, we present solutions to prevent Denial-of-Service (DoS) attacks that misuse RA [65]. For large-scale systems, or swarms of connected devices, efficient RA protocols are required that perform significantly better than attesting each device individually. We present the first collective attestation scheme, called SEDA [34], that allows the efficient attestation of very large swarms of devices. In autonomous systems, where embedded devices take both roles, prover and verifier, advanced attestation schemes, such as run-time attestation [3], are not applicable, due to the limited resources of these devices. We overcome this central limitation and enable run-time attestation for collaborative embedded systems with our Data Integrity Attestation (DIAT) [4]. DIAT's presents a novel attestation paradigm, as it proves correctness of exchanged data rather than the correctness of entire systems.

1.1 Contributions

In this work we make contributions in the area of Trusted Execution Environment (TEE) architectures and their applications. Furthermore, we introduce novel and extended Remote Attestation (RA) schemes.

- We design and develop two new security architectures.
  - We present TyTAN, the first security architecture for low-end embedded systems that enables dynamically managed, mutually distrusting TEEs while ensuring real-time compliance of the overall platform [62].
  - Sanctuary is our novel security architecture for mobile systems that brings enclave-like TEEs to ARM TrustZone devices [73].
- We investigate side-channel attacks and defenses for Intel's Software Guard Extensions (SGX).
  - We show a novel and powerful cache side-channel attack against SGX leveraging the adversary's amplified capabilities and control over the target system in the context of SGX [68].
  - We develop HardIDX, a database search index leveraging SGX to securely search outsourced databases while preventing information leakage due to data accesses to untrusted memory outside of the HardIDX-enclave [149, 151].
  - DR.SGX is our fully automated side-channel defense mechanism for SGX, based on a novel concept, called semantic-agnostic data randomization [72]. DR.SGX's fine-granular data randomization thwarts an adversary from in-

ferring sensitive information from memory access patterns that observable through various side channels.

- We present novel applications that leverage TEEs to allow privacy-preserving speech recognition as well as policy enforcement for smart devices.
  - Our Automated Speech Recognition (ASR) framework VoiceGuard uses SGX to protect both, the user's voice input data, i. e., user's privacy, as well as the speech recognition model, i. e., the provider's Intellectual Property (IP) [71].
  - Based on ARM TrustZone, we develop a novel method to remotely enforce usage policies on smart devices while asserting the device owner's security and privacy [64].
- We improve and advance RA beyond the classical one-to-one attestation scenarios, where a trusted verifier validates the state of a single prover device.
  - We develop solutions to extend RA protocols to protect against malicious verifiers that try to misuse RA to attack prover devices [65].
  - We introduce SEDA [34], the first collective attestation scheme for efficient verification of swarms of devices, inspiring many follow-up works [16, 199, 201, 82].
  - We devise a novel attestation concept that enables efficient and secure interaction of collaborative devices, called data integrity attestation (DIAT) [4]. It presents a paradigm shift in the area of RA, moving away from attestation of entire device to ensure their integrity towards ensuring the integrity of generated and processed data.

1.2 Previous Publications

This work is based on the following peer-reviewed publications, which are structured into four chapters: Chapter 3 to Chapter 6; the full list of the author's publications is provided in Chapter 8.

Chapter 3: Security architectures TyTAN and Sanctuary.

Ferdinand Brasser, Patrick Koeberl, Brahim El Mahjoub, Ahmad-Reza Sadeghi, and Christian Wachsmann. TyTAN: Tiny Trust Anchor for Tiny Devices. In *Proceedings of IEEE/ACM Design Automation Conference* (DAC), 2015.

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

Chapter 4: Side-channel attack on Software Guard Extensions (SGX) enclaves, SGX-secured database index HardIDX and side-channel defense through data randomization with DR.SGX.

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

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

Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In *Proceedings of Annual Computer Security Applications Conference* (ACSAC), 2019.

Chapter 5: Restricted spaces policies enforcement on mobile devices and private speech recognition with SGX.

Ferdinand Brasser, Vinod Ganapathy, Liviu Iftode, Daeyoung Kim, Christopher Liebchen, and Ahmad-Reza Sadeghi. Regulating ARM TrustZone Devices in Restricted Spaces. In *Proceedings of International Conference on Mobile Systems, Applications, and Services* (MobiSys), 2016.

Ferdinand Brasser, Tommaso Frassetto, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, and Christian Weinert. VoiceGuard: Secure and Private Speech Processing. In *Proceedings of Interspeech*, 2018.

Chapter 6: Verifier authentication in Remote Attestation (RA), swarm attestation SEDA and data integrity attestation DIAT.

Ferdinand Brasser, Kasper Rasmussen, Ahmad-Reza Sadeghi, and Gene Tsudik. Remote Attestation for Low-End Embedded Devices: the Prover's Perspective. In *Proceedings of IEEE/ACM Design Automation Conference* (DAC), 2016.

N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. SEDA: Scalable Embedded Device Attestation. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security* (CCS), 2015.

Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. DIAT: Data Integrity Attestation for Resilient Collaboration of

Autonomous Systems. In *Proceedings of Annual Network and Distributed System Security Symposium* (NDSS), 2019.

2 PRELIMINARIES AND BACKGROUND

In this chapter we provide general background relevant for this work. Background specific to individual system and solutions is provided in the corresponding section.

2.1 Computer Architectures

Most computer systems follow a hierarchical protection and management paradigm. Software entities can execute with different privileges, where the privileges of an entity residing higher in the hierarchy are a super-set of the privileges of entities residing lower in the hierarchy. For instance, the Operating System (OS), as the higher privileged entity, has accesses to its own memory and the memory of applications or processes, which are less privileged, while, processes cannot access the memory of the OS. This is enforced via memory access control that is under the OS's governance (cf. Section 2.1.1). This principal repeats at multiple layers, as illustrated in Figure 1.

Similarly, (unprivileged) processes have access only to a subset of Central Processing Unit (CPU) instructions, while additional instructions are only available to privileged software entities, such as the OS. Typically, instructions that are used to configure the system and its resources, e. g., define access rules, are reserved for more privileged entities.

The CPU can operate in different modes defining the privilege levels of the software currently in execution.[1] The naming of the CPU modes is platform specific. For instance, on ARM processors the modes are called exception levels and are numbered from lowest privileges (`EL0`) to highest privileges (`EL3`). On x86, in contrast, unprivileged code is executed in so-called `ring 3`, while the OS executes in `ring 0`. Following this systematic, higher privileged modes are numbered in descending order, for instance, the Virtual Machine Monitor (VMM) mode is often referred to as "`ring -1`".

Transition between different CPU modes can be initiated by different events, such as exceptions or interrupts. Also, external events, for instance hardware interrupts, cause a transition to privileged execution mode where interrupts are handled by Interrupt Service Routines (ISRs). Unprivileged entities can also initiate a transition. Typically, this is to request services from the OS, via system calls, by executing dedicated instructions, for example `sysenter` or `syscall` on x86 systems. The OS can return to unprivileged entities using corresponding instructions, e. g., `sysexit` or `sysret`.

The higher privileged software is responsible for managing, i. e., controlling resource access, as well as providing services for less privileged software. When the less privileged software is aware of its limitations and the existence of a higher privileged entity, then it

1 On multi-core systems each CPU-core can operate in different modes independently.

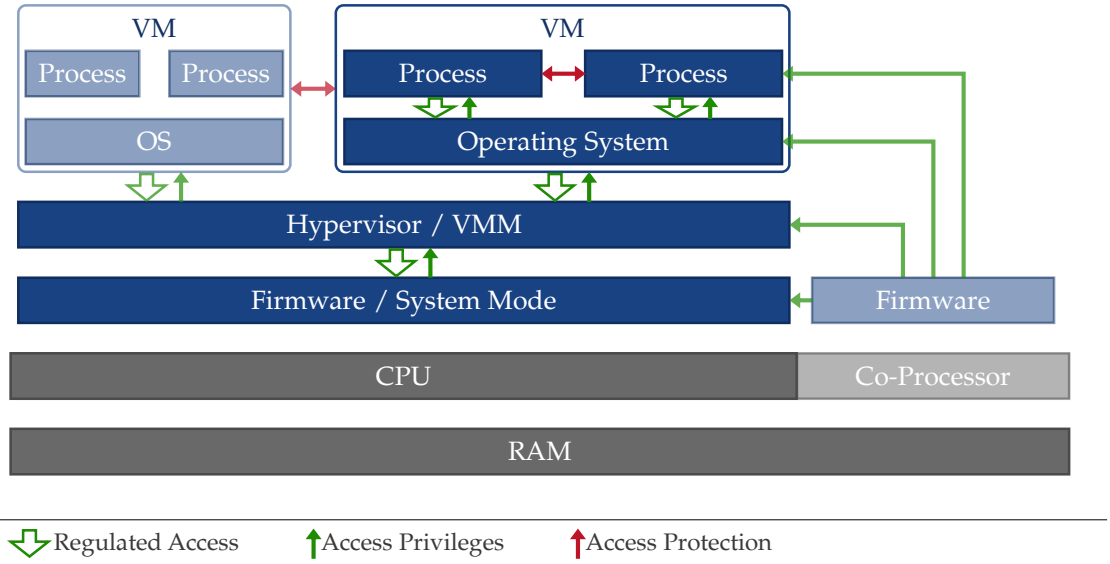


Figure 1: Common computer architecture using a hierarchical protection and management paradigm. Higher privileged entities can access unprivileged entities (unrestricted), while unprivileged entities have no direct access to privileged entities.

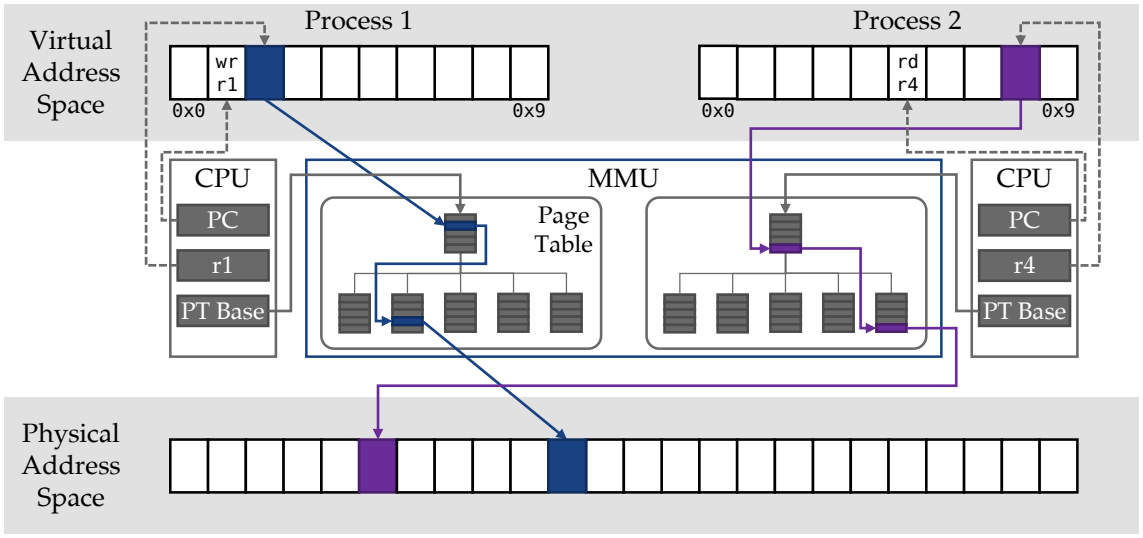
explicitly requests services from the higher privileged entity, e. g., an application requests services from the OS through system calls, invoking a predefined function of the OS. Otherwise, the higher privileged entity has to be transparent and needs to emulate the behavior of the underlying system as expected by the less privileged entity. For instance, hypervisors can provide the OS executing in a Virtual Machine (VM) the impression of running directly and exclusively on the computer's hardware.

While the main CPU and the software executing on it adhere to this hierarchical model, many platforms have additional system components that are orthogonal to this model. For instance, Intel's CSME is a co-processor that can access the main memory enabling it to access the data and code of all software entities, regardless of the privilege level [148], as illustrated in Figure 1.

2.1.1 *Memory Management and Memory Protection*

Memory access control is a central concept for computer security. Limiting the access to memory allows the protection of information's integrity and confidentiality stored in it. In general, potential malicious or faulty entities should be restricted with respect to the memory locations that they can access. Additionally, for accessible memory location the mode of access should be regulated, e. g., allowing read-only access or execute-only access.

Memory access control can be realized using different approaches, e. g., by testing memory accesses for compliance with a given list of rules or by encrypting the memory content. Subsequently techniques relevant for the rest of this work are described.



PC: Program Counter PT: Page Table r1/4: Register 1/4 rd: read wr: write

Figure 2: The Memory Management Unit (MMU) translates virtual memory addresses to physical memory addresses. The mapping between virtual memory and physical memory is defined per process and is managed via a hierarchical data structure, called page tables. The root of the currently executing process' page table is stored in a dedicated CPU-register (PT Base).

Read-Only Memory. Memory that cannot be written to, typically due to its physical properties, provides strong integrity protection of its content. Alternatively, Read-Only Memory (ROM), i. e., memory must not be written to, can be implemented using approaches that enforce access permissions for memory, e. g., using a Memory Management Unit (MMU) or Memory Protection Unit (MPU).

Memory Management Unit. The concept of *virtual memory* provides software entities, in particular processes, with a continuous memory space independent of other software entities running on a system. This is achieved by translating all memory access of a process: processes operate on virtual memory addresses which are mapped and translated on the fly to addresses in the computer's physical memory. Each process has its own virtual memory space, which gets mapped to physical memory locations exclusively reserved to each process. This mapping is managed by privileged software, typically the OS.

Figure 2 shows the virtual address space of two processes. The mapping of virtual memory to physical memory is done in fixed units of memory, called memory pages. For each memory page of the virtual memory space, its mapping to the corresponding memory page in physical memory is stored in a hierarchical data structure, called page table. And for each virtual memory space a separate page table maps virtual memory pages to physical memory pages.

The Memory Management Unit (MMU) is a hardware unit that translates all memory accesses, i. e., for each memory access the MMU substitutes the virtual memory address

with the corresponding physical memory address using the page table, as shown in Figure 2. The page table to be used by the MMU is specified by a CPU-register. Thus, the active virtual address space can be defined by updating this register, e. g., the OS updates it when scheduling a new process.

The page tables define for each memory page access permissions, i. e., the access mode for each memory page. Additionally, not all virtual memory pages need to be mapped to physical memory addresses, virtual pages for which no mapping exist are inaccessible, and thus, providing isolation. For instance, physical memory pages that are mapped to one process' virtual memory are inaccessible for all other processes that do not have a mapping from their virtual memory space to these physical memory pages. When software tries to access a virtual memory page that is not mapped to a physical page, an exception is raised by the MMU notifying the OS about the access attempt.

Swapping. Not having physical memory pages mapped to the virtual memory space of a process is also done for dynamic resource management, allowing the OS to over-provision the system's physical memory. The OS can mark virtual memory pages as unavailable, resulting in an exception when the process tries to access that page. During exception handling, the OS can allocate a physical memory page, create a mapping for the virtual memory page that caused the exception and make the virtual memory page as available. Afterwards, the process can continue, using the previous unavailable virtual memory page.

When no physical memory is available, the OS has to re-use a physical memory page that is already allocated and mapped to another virtual memory page, i. e., *swap* the memory pages. To prevent data loss, the OS copies the content of the memory page that should be freed to another memory system, e. g., non-volatile memory. When the content of the swapped out memory page is again accessed by its process, the OS has to revert the process, i. e., restore the page's content to a newly allocated physical memory page and map the process' virtual memory page to the new physical memory page.

Memory Protection Unit. The MPU is a hardware unit that provides memory protection by enforcing memory access control policies. It is most commonly used in low-end embedded systems that do not provide virtual memory abstraction, i. e., where all software operates in the physical address space.

The MPU moderates all accesses of the CPU to the memory, as shown in Figure 3. The access control policies can be defined per memory region, i. e., different parts of the memory can be accessed in different modes. Thus, regions can be marked non-executable or can be marked non-writable.

The memory access rules have global validity, i. e., *all* rules are enforced regardless of the software entity currently executing. On systems with different execution modes, e. g., privileged and unprivileged mode, often distinct rules for each mode can be defined, e. g., allowing to isolate privileged code and data from unprivileged software's accesses.

Execution-Aware Memory Protection Unit. The EA-MPU extends the concept of MPUs by making it *execution-aware*, i. e., it takes the current executing software into account when deciding whether a memory accesses should be allowed or denied [226]. Access permission rules get extended by a second address range, which defines the code region

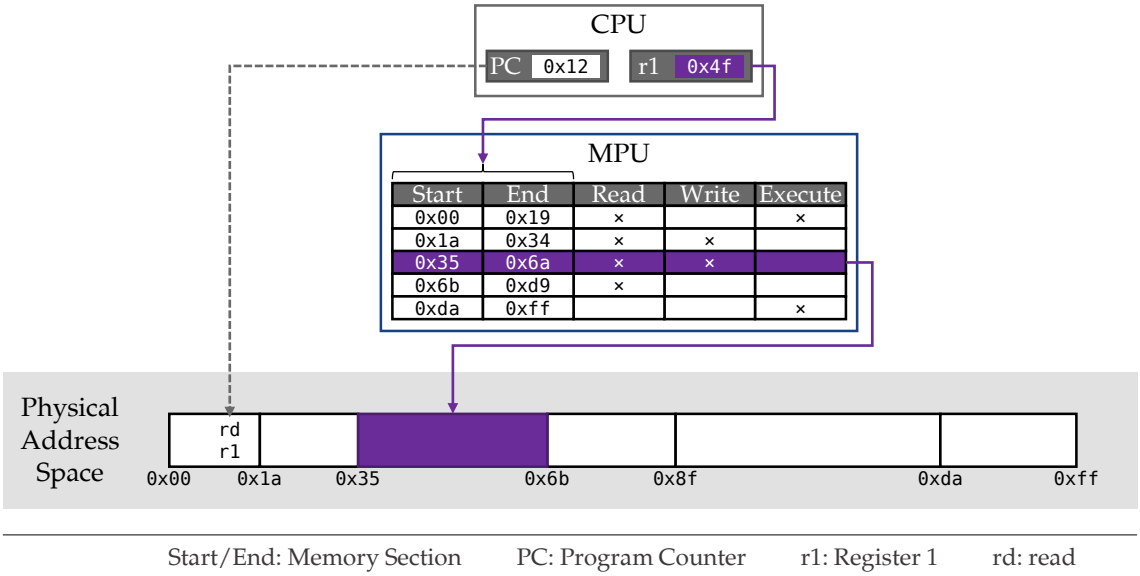


Figure 3: The Memory Protection Unit (MPU) enforces memory access rules, which are defined per memory regions.

for which an access rule is valid. On each memory access, the EA-MPU checks whether the currently executing code, identified by the Program Counter (PC) register, lies in the specified code region, in addition to the access mode checks performed by a standard MPU, as shown in Figure 4.

This concept allows memory access rules to be defined per software entity, e. g., process or task, independent of privilege levels.

2.1.2 *Cache Architectures*

In the following we provide details of the Intel x86 cache architecture [208, 205].[2] We focus on the Intel Skylake processor generation, i. e., the type of CPU we used for our side-channel attack and defense presented in Chapter 4.

Memory caching "hides" the latency of memory accesses to the system's Dynamic Random Access Memory (DRAM) by keeping a copy of currently processed data in faster memory, called cache. Caches are fast but small memories, which are built into the CPU. Typically, they are realized as Static Random Access Memory (SRAM). Although SRAM caches have better performance compared to DRAM memory, they cannot be produced at the same density as DRAM, resulting in higher cost. Due to their higher cost (e. g., in terms of production and energy consumption), caches are orders of magnitude smaller than DRAM and only a subset of a system's memory content can be present in the cache at any point in time, i. e., to leverage their advantages of both memory types they are used in combination.

2 We will use the terminology from Intel documents [203].

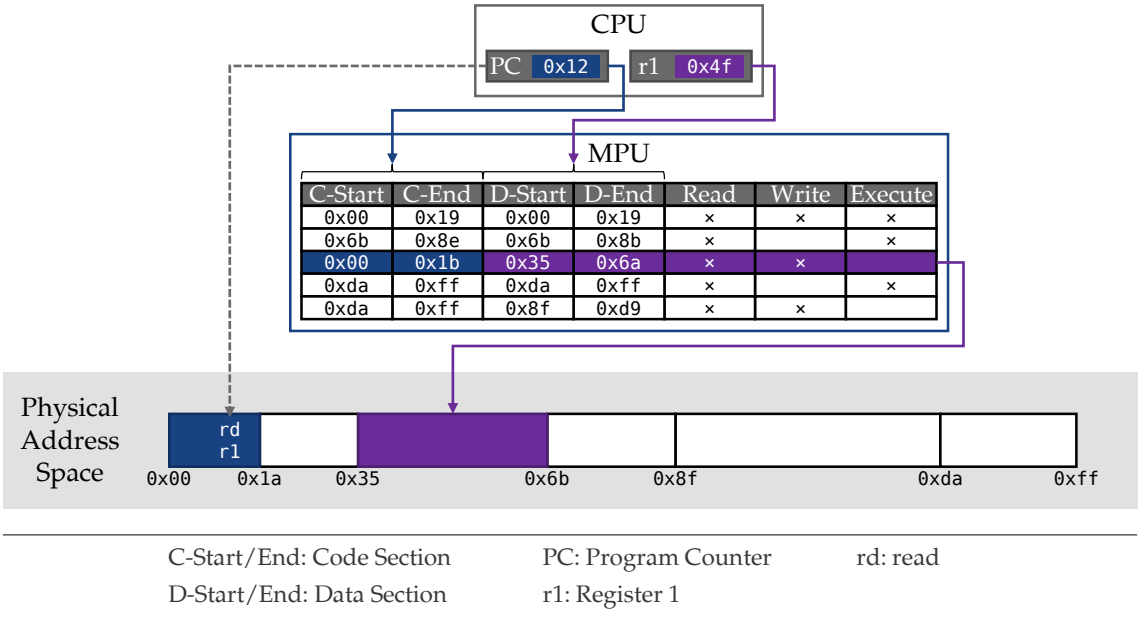


Figure 4: The Execution-Aware Memory Protection Unit (EA-MPU) enforces memory access rules depending on the currently executing software, i. e., memory access is only granted to specified software entities, which are identified based on the code region currently being executed.

When a memory operation is performed, the cache controller checks whether the requested data is already cached, and if so, the request is served from the cache, called a *cache hit*, otherwise *cache miss*.[3]

The cache controller aims to maximize the cache hit rate by predicting which data are used next by the CPU. This prediction is based on the assumption of temporal and spatial locality of memory accesses. Temporal locality means that recently used data are more likely to be used soon than data that have not be accessed in a while. Spatial locality describes the assumption that data located next to data currently in use is likely to be used as well. Hence, two types of data should be available in the cache: data that have been used most recently, and data close to those data. The first is accounted for by the cache replacement strategy. The latter is achieved by loading chunks of data into the cache, called *cache line*.

For each memory access the cache controller has to check if the data are present in the cache. Sequentially iterating through the entire cache would be very time-consuming. Therefore, the cache is divided into *cache lines* and for each memory address the corresponding cache line can be quickly determined, typically the lower bits of a memory address select the cache line. Hence, multiple memory addresses map to the same cache line. In Figure 5 the first line of each *cache page* in memory maps to the first cache line.[4]

3 For the rest of this work we focus on read operations from data caches. Also, we exclude *uncacheable* memory from our discussion, i. e., memory regions that are explicitly prevented from being stored in the cache, which is, for instance, required for ensuring coherency with external devices supporting Direct Memory Access (DMA).

4 In general, *cache page* are different from virtual memory pages.

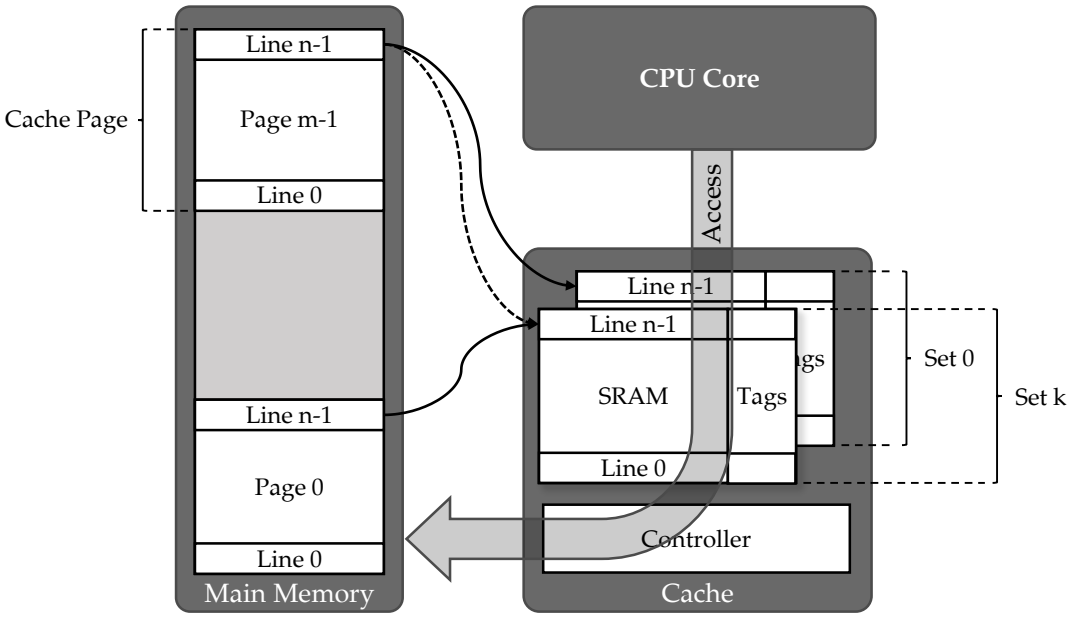


Figure 5: Caches are organized in ways and sets. The cache way is selected based on (a part) of the memory address of the data to be cached. The sets refer to the number of concurrent cache locations data can be stored.

Having one cache entry per cache line quickly leads to conflicts and the controller has to evict data from cache to replace it with newly requested data. For instance, in Figure 5 the first line of cache pages 0 and $m-1$ are mapped to the same cache line. To minimize such conflicts caches are often (set) associative. Multiple copies of each cache line exist in parallel, also known as *cache sets*, thus #cachesets many data from conflicting memory locations can stay in the cache simultaneously. When the maximum number of allowed conflicts is exceeded, the cache controller must evict data from a cache line to replace it with newly requested data. Which cache line to evict is determined by the cache replacement policy. The "least recently used" policy is often used, i. e., the data that has not been access for the longest time frame is evicted.

Figure 6 shows the cache hierarchy of current Intel CPUs, with a three level hierarchy of caches: The Last Level Cache (LLC), also known as Level 3 (L3) cache, is the largest and slowest cache; it is shared between all CPU-cores. The Level 1 (L1) and Level (L2) caches are exclusive to each CPU-core, i. e., each CPU-core has a dedicated L1 and L2 cache. The L1/L2 caches are shared between Simultaneous Multithreading (SMT) execution units.[5]

The LLC is inclusive, i. e., all data that is present in the L1 or L2 cache of any CPU-core is also present in the LLC, as shown for *Core* 0 in Figure 6. L2 cache and the LLC are physically indexed, while the L1 cache is virtually indexed. This means that the L1 cache is accessed and managed based on the virtual memory address as used by the applications directly. When accessing the other caches, virtual addresses first get translated to physical addresses to determine the cache line. Another unique feature of the L1 cache is its separation into data and instruction cache. Code fetches only affect the

5 SMT is also known as Hyper-threading (HT) in Intel CPUs.

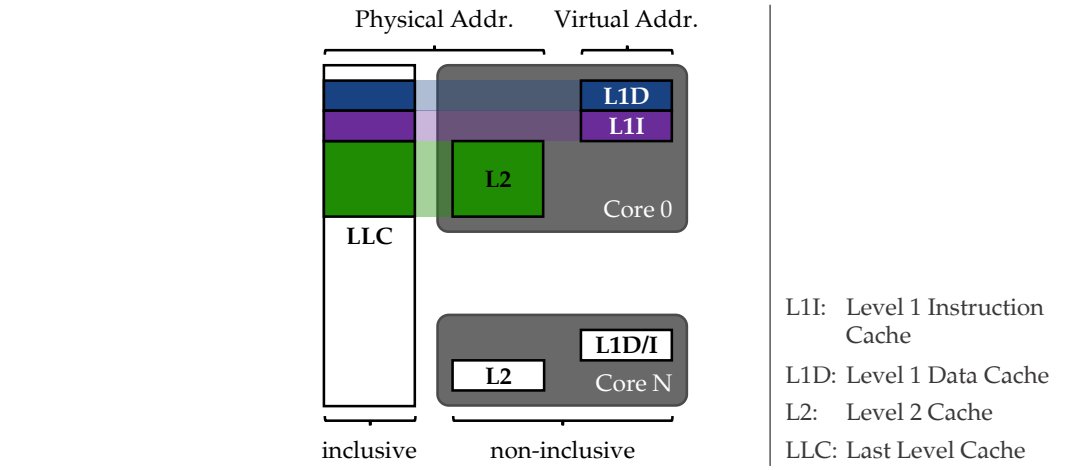


Figure 6: Cache hierarchy and configuration of Intel Skylake processors. The Last Level Cache (LLC) is inclusive, i. e., all data stored in any per-core L1/L2 is also stored in LLC. L1 cache is divided into separated parts for data and instructions. The L1 cache is addressed using virtual addresses, while L2 and L3 caches are addressed by physical addresses.

instruction cache and leave the data cache unmodified, and vice versa. In L2 and LLC caches code memory and data memory compete for the available cache space.

2.2 Security Services

2.2.1 *Secure Boot*

Secure boot gradually verifies the integrity of a system's software components while it is starting. Before any component is executed its integrity is checked. The verification process is started by the initial component E_0 of the platform's boot process (e. g., the BIOS or UEFI code) and iteratively continued by all components $E_1 \ldots E_n$ that are executed afterwards (e. g., the boot loader, hypervisor, and the Virtual Machines (VMs)) until the last component E_{n+1} has been verified and executed (cf. Figure 7).

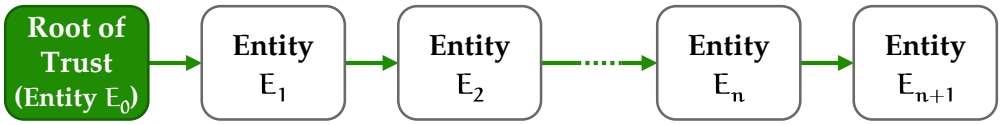


Figure 7: Chain of Trust (CoT) concept.

Since the integrity of each component E_i is verified by its predecessor E_{i-1} before it is executed, only known (i. e., authentic) software components are loaded and executed. When the integrity of a software component cannot be verified, different reactions are possible. For instance, the system may stop execution (also called fail secure mode) or it may use an authentic fallback version of the software component whose integrity check

failed [23]. The integrity verification of a component is meaningful only if the component that performs the verification itself is benign. Hence, the initial component E_0 must be trusted and is often denoted as Root of Trust (RoT). This means that the integrity of E_0 must be protected against (software) attacks, e. g., by storing the code and data of E_0 in Read-Only Memory (ROM). The integrity of all other components $E_1 \dots E_n$ is ensured by the fact that the integrity of every component E_i is verified by its predecessor E_{i-1} before E_i is executed.

The method used to verify the integrity of a component depends on the requirements of the application. The most common approach is for the secure boot mechanism to compute an Integrity Measurement Value (IMV), which typically is the cryptographic hash digest, of the binary code of the software component to be verified. The IMV is then compared to a reference IMV that is typically certified by the platform manufacturer, the platform user, or the software provider. If the IMV computed by the secure boot mechanism matches the certified reference IMV provided by the software provider, the integrity of the software component is preserved.

2.2.1.1 *Order of Integrity Verification*

It must be ensured that the integrity of each software component is verified before it is executed. Except for this fundamental rule, there are no other restrictions on the order of the integrity verification and software execution. This means that the integrity of the software component E_n can be verified by any software component $E_0 \dots E_{n-1}$ that is executed before E_n.

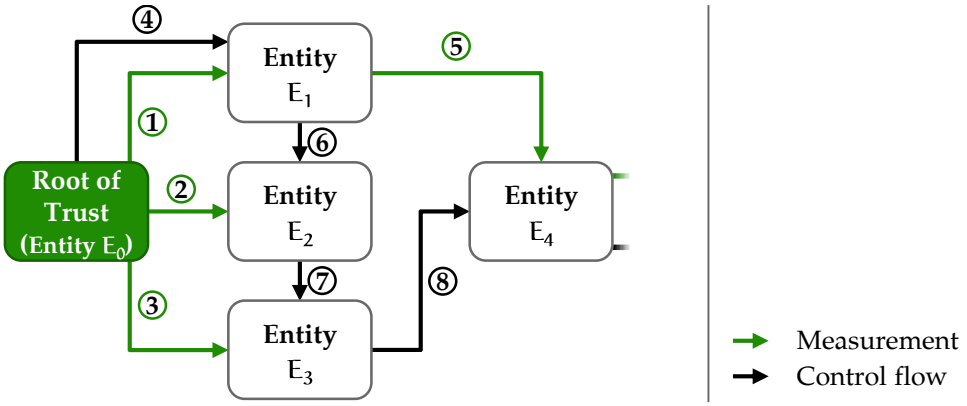


Figure 8: Integrity validation and execution order in Chain of Trust (CoT).

Figure 8 illustrates such a scenario, where the integrity of software components E_1, E_2, and E_3 is verified by software component E_0 (①, ② and ③ in Figure 8) before E_1, E_2, or E_3 is executed (④, ⑥ and ⑦ in Figure 8). Similarly, E_1 verifies E_4 (⑤ in Figure 8) although E_1 is not loaded directly before E_4. Nevertheless, in this example, the integrity of each software component is verified before it is loaded.

2.2.1.2 *Reference Measurement Values*

The integrity and authenticity of the reference IMV must be ensured. Depending on the requirements on the flexibility of the secure boot mechanism, different approaches are possible to store and manage reference IMVs.

Reference IMVs Embedded in Software Components. One approach to manage reference IMVs is to embed the reference IMV of software component E_i into the predecessor component E_{i-1} that verifies the integrity of E_i, as illustrated in Figure 9. The initial component E_0 includes the reference IMV of E_1 denoted $M(E_1)$, i. e., the expected IMV of E_1. The integrity of $M(E_1)$ is protected by the same mechanism that ensures the integrity of E_0 itself. The integrity of the actual software component E_1' is verified by E_0 before E_1' is executed. More detailed, E_0 computes the hash digest of the binary code of E_1', denoted $M(E_1')$ (① in Figure 9) and compares the result with the reference IMV $M(E_1)$ stored in E_0 (② in Figure 9). The verification of the integrity of E_1' succeeds only if $M(E_1')$ matches $M(E_1)$. If this is the case, E_1' is executed (③ in Figure 9) and takes over the role of E_0, i. e., E_1' measures the binary code of E_2' and executes E_2' only if its measurement $M(E_2')$ matches the reference IMV $M(E_2)$ of software component E_2, which is stored in E_1'. This process is continued until the integrity of all software components has been verified.

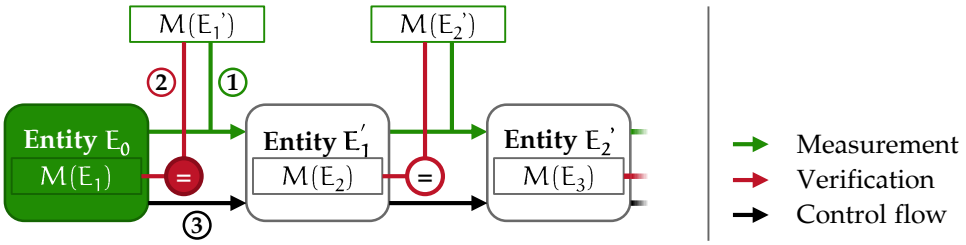


Figure 9: Secure boot with embedded IMVs.

One major limitation of this approach is its lack of flexibility. Specifically, updating software component E_i requires updating the reference IMVs in all software components E_j with $j < i$.

Reference IMVs Managed in Central Database. An more flexible approach, compared to the embedding approach described before, is managing the reference IMVs in a central database, illustrated in Figure 10. Before the initial component E_0 passes execution to component E_1', it verifies the integrity of E_1'. Again, E_0 measures the binary code of E_1' and compares the result $M(E_1')$ with the reference IMV $M(E_1)$ of E_1. However, this time the reference IMV is stored in a central database that can be read by all software components performing integrity verification. To ensure the authenticity and integrity of the reference IMVs, the integrity of the database must be protected. One approach to protect the integrity of the database is using the same method used to protect the integrity of the initial component E_0 or to use E_0 to verify the integrity of the database.

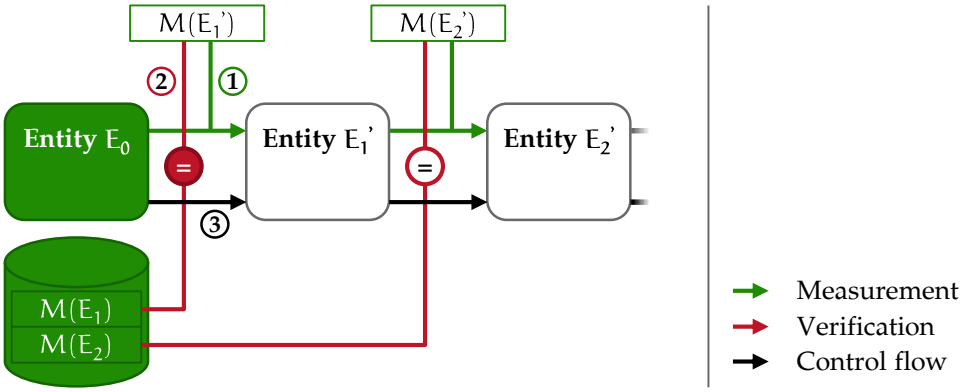


Figure 10: Secure boot with central IMVs storage.

This method allows the flexible update of individual software components. However, updating software components requires updating the corresponding reference IMVs in the database in an authentic way.

Certified Reference IMVs. The most practical approach to implement secure boot to is using digital signatures (certificates) to ensure the integrity and authenticity of reference IMVs. This approach is shown in Figure 11. The reference IMV of every software component is contained in a digital certificate issued by a signing authority. The signing authority might be the platform manufacturer, platform user, and/or a software provider. The certificate does not need to be stored in protected memory since its authenticity is ensured by a digital signature σ_{pk} issued by the signing authority. Before the initial component E_0 passes execution to E_1', it verifies the integrity of E_1'. Again, E_0 measures the binary files of E_1' and compares $M(E_1')$ to the reference IMV $M(E_1)$ of E_1', which is included in the certificate. The authenticity of the reference IMV is checked by verifying the certificate using the authentic public verification key pk of the signing authority.

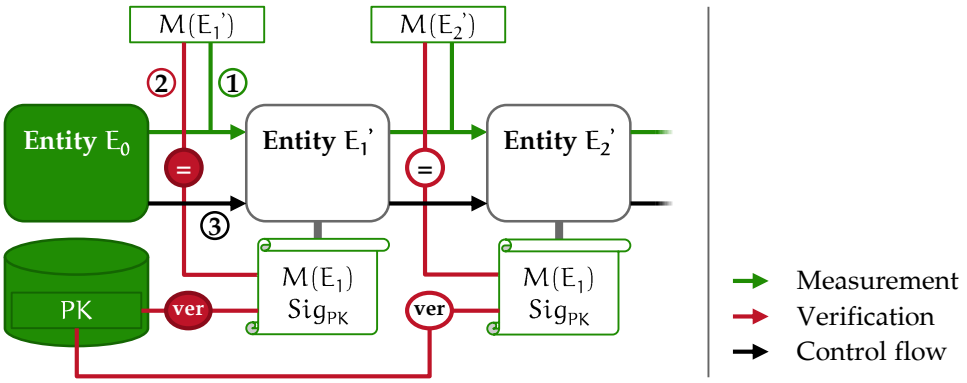


Figure 11: Secure boot with IMV certificates.

When updating a component E_n, a new certificate must be issued for the version of E_n and stored on the platform. Since the certificate can be validated with the same public verification key pk, neither entity E_0 nor the protected memory containing pk must be

updated. Since E_0 and pk need to be updated rarely or even never during the lifetime of the platform, simple hardware-based protection such as ROM can be used to protect their integrity and authenticity.

Revocation. In secure boot systems, revocation might be necessary either because an entity E_n is no longer trustworthy and should not be allowed to be executed, e. g., because a vulnerability was discovered in its code and an update version E_n^* has been released, or a signing key, used to authenticate code certificates, was compromised.

In the first case, the measurement $M(E_i)$ must be removed and replaced with an updated measurement $M(E_i^*)$. Depending on the secure boot variant used, this requires updating E_i's predecessor components with an updated embedded IMVs ($M(E_i^*)$), updating the central IMV storage or issuing a new certificate for $M(E_i^*)$. In all variants the authenticity of the updates must be ensured.

In the second case, the secure boot system must learn that the compromised key can no longer assert the integrity of any entity E_n, or the authenticity and integrity of other keys, e. g., when a Public Key Infrastructure (PKI) is used in the secure boot system.

In all cases the revocation information must be made available to the secure boot system. If a device is not compromised, it can retrieve the updated information over the network, e. g., in form of a Certificate Revocation List (CRL) to revoke certificates [105]. A device under an adversary's control, e. g., a device on which the adversary exploited a vulnerability in an entity E_i, will not voluntarily update the revocation information. In these cases, the update must be enforced, for instance, certificates can be issued with an expiration date rendering updates mandatory after the certificate validity period. However, these solutions require that the secure boot system can receive updated information, e. g., via network. Further, to prevent attacks, such as roll-back attacks, devices need to store the latest revocation information securely or maintain version information using a secure monotonic counter. To ensure up-to-date revocation information, devices need access to reliable time information.

If a secure boot system's RoT is compromised, e. g., if E_0 is insecure, the RoT must be replaced or updated [289], which is impossible in many systems.

2.2.2 *Attestation*

Attestation enables one entity, called *verifier*, to gain assurance about the state of another entity, called *prover*. The rationale behind attestation is to verify that the prover is in a correct state and is therefore trustworthy, i. e., attestation is often used to bootstrap trust in the prover.

Typically, the state of a device is defined by its memory content. In particular, the memory that is determining the behavior of a computing device is considered, i. e., memory containing the provider's program code, such as binary code loaded in Random Access Memory (RAM). This type of attestation is also called *binary attestation* or *static attestation*, as it captures the static state of the prover, allowing the detection of malware infections and other attacks that alter the code memory of a prover device [382, 139]. More sophisticated attack techniques, for instance Return-Oriented Programming (ROP)

attacks, can only be captured by approaches that record and report the run-time behavior of the prover device, called *run-time attestation* [3, 130, 129, 416].

The correct state of a prover device can be defined arbitrarily, however, in practice the state is typically considered correct if it conforms with the manufacturer's specifications for the prover device, i. e., if the state has not been altered in an unauthorized way.[6]

Most attestation mechanism represent the prover's state in a compressed form, typically, the state is fed into a cryptographic hash function to map it to a short, fixed size value. The verifier usually is assumed to know all hash values representing correct states, hence, the verifier can decide, based on the value provided by the prover, whether the prover is in a correct state.

For the verifier, to be able to establish trust in a prover device based on the reported state, the following necessary conditions must hold. (1) binding between state measurement value and prover authentic identity, (2) integrity of state measurement value, (3) freshness of state measurements, (4) trustworthy state capture mechanism, (5) unpredictable time of attestation, i. e., the time of attestation must be unknown in advanced to a potential adversary, and (6) equality of the prover's state and its measurement regarding prover's correctness.

In different systems and scenarios these requirements can be assumed or can be met by different means.

Local attestation scenarios rely on a trusted link or channel between prover and verifier, e. g., a connection via a dedicated physical wire which is assumed to be untampered. Hence, the authenticity of the prover entity and the integrity of the transmitted measurement value can be assumed. In Remote Attestation (RA) schemes, the authenticity of the prover and the integrity of the transmitted measurement value must be ensured by other means. This can be done, for instance, using a digital signature or Message Authentication Code (MAC), however, the key used to authenticate the prover must be protected such that it is only accessible to trusted entities and each prover needs to have a unique cryptographic key. Additionally, the signature/MAC binds the measurement value to the prover's identity.

The freshness of a state measurement can be achieved by integrating a nonce in the measurement that cannot be foreseen by a potential adversary and for which the verifier can validate that it is timely, otherwise the adversary can pre-compute a valid attestation report or reuse a previous report. Either the verifier sends a fresh nonce to the prover, i. e., the verifier knows when the nonce was disclosed to the prover, or a nonce is generated in a trustworthy way on the prover device. If the verifier initiates the attestation process, it can follow a strategy unknown to the adversary. However, if the prover reports its state independent of verifier requests it is important that an adversary can neither influence this process, e. g., prevent or delay the reporting, nor foresee the attestation time, i. e., allowing a *roaming adversary* to leave the prover device reverting all its changes before attestation is performed (cf. Section 6.1.1.2).

The measurement reported to the verifier must be correct, i. e., an adversary must not be able to tamper with the measurement process. In particular, the integrity of

6 Authorized alterations of a device's software, e. g., due to a software update, usually should not lead to an incorrect state.

the component performing the measurement must be ensured, otherwise the verifier cannot trust the attestation report. Additionally, the temporal integrity of the measured state must be ensured while the measurement is being performed [84]. The integrity of the measurement component is achieved in different ways by different RA schemes. Existing RA approaches can be divided into three categories: (i) software-based attestation, (ii) hardware-based attestation, and (iii) hybrid software/hardware-bases attestation. In hardware-based and hybrid attestation solutions the measurements engine's integrity is ensured, e. g., by isolating it in an isolated execution environment such as a co-processor or in a Trusted Execution Environment (TEE). The same protection mechanism can also be used to protect cryptographic keys to authenticate the prover to a remote verifier and protect the integrity of an attestation report when transmitted via network. Software-based attestationrelies only on unprotected software to measure the state of the prover device. Hence, the integrity of the measurement engine cannot be guaranteed. The integrity of the measurement engine is tested using its execution characteristics, such as its timing behavior. The assumption is that the measurement engine is time-optimal, i. e., no faster way of measuring the prover's state exists, thus, if an adversary manipulated the measurement engine, its run time increases, which will be detected by the verifier. However, these timing assumptions only hold if the adversary is limited to the prover device's unaltered resources for generating an attestation report. If the adversary, for instance, increases the processing speed of the prover or receives assistance in the computations from another device, these underlying assumptions do not hold, and hence, software-based attestation cannot provide the required security guarantees. Furthermore, the lack of protection capability averts the usage of cryptographic methods to authenticate and protect the integrity of attestation reports, as the required cryptographic keys cannot be protected from unauthorized access by the adversary.

2.3 Security Architectures and Trusted Execution Environments

2.3.1 *ARM TrustZone*

TrustZone represents a set of security enhancements to ARM's processor and System-on-Chip (SoC) designs. TrustZone extends the processor, memory system (including caches), and peripherals. A TrustZone-enabled processor can execute in two security modes at any given time. The two modes are called *normal world* and *secure world*, respectively. The normal world retains backwards-compatibility and hosts all software that is not explicitly made for the secure world. All security critical software is meant to be isolated in the secure world, protected from the untrusted software in the normal world. The secure and normal world both manage their own address spaces using the traditional privilege levels for separation of the Operating System (OS) kernel and application code (cf. Section 2.1). More specifically, current ARM processors provide four privilege levels (*Exception Levels* – `EL0` to `EL3`). The normal world provides `EL0` (unprivileged applications), `EL1` (privileged OS), and `EL2` (hypervisor, also privileged). In the secure world, two privilege levels (`S-EL0`

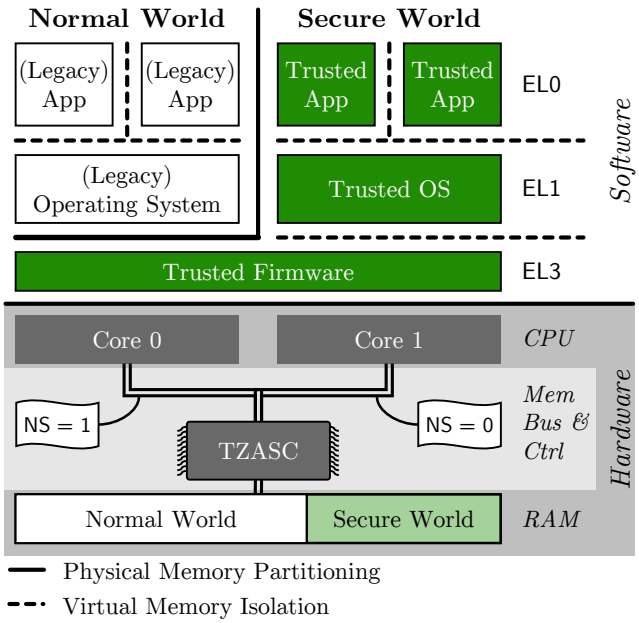


Figure 12: TrustZone software and hardware components. Software can be executed in *normal world* or in *secure world*. Isolation between these two worlds is enforced by the memory controller (TZASC) that checks for each memory access which world it originates from.

and `S-EL1`) are available in current ARM architectures.[7] The highest privileged mode `EL3`, also called *monitor mode*, executes the ARM Trusted Firmware (TF).

Typically, the normal world hosts the user facing software, including a smartphone OS, e. g., Android [172], and applications (often abbreviated "apps" in the context of smartphones). In the secure world a Trusted OS (TOS) is running that manages the Trusted Apps (TAs), see Figure 12.

Each processor core can switch from normal to secure world via a dedicated secure monitor call (`smc`) instruction. When an `smc` instruction is invoked from normal world, the processor core performs a context switch to the secure world (via the monitor mode), the execution of the normal world is suspended while the secure world is executing. In a multi-core system, each processor core can independently execute in any of the modes, i. e., at any privilege level (`EL0` to `EL3`) in either world.

TrustZone can separate physical memory into two partitions, with one partition being exclusively accessible by the secure world. This isolation is enforced by the memory controller (TZASC) that evaluates for every memory transaction whether it originates from the secure world or the normal world. This evaluation is done based on an addition flag included in each transaction, called the non-secure flag (`NS`), indicating that the transaction was issued by a processor in normal mode (normal world, $\texttt{NS} = 1$) or in secure mode (secure world, $\texttt{NS} = 0$). Accesses to the memory partition reserved for the secure world is only granted if the processor issuing the access is executed in secure-

7 Future versions of the ARM architecture will support `S-EL2`, i. e., hardware virtualization in the secure world [31].

world mode (or monitor mode / EL3). While the normal world cannot access memory assigned to the secure world, the secure world can access normal-world memory.

Further, TrustZone allows individual peripheral devices to be assigned exclusively to the secure world. For these peripherals, hardware interrupts are directly routed to and handled by the secure world. Accesses to the devices configuration interface, which occur via transactions on the system bus, are restricted based on the non-secure flag (NS), i. e., only bus transactions with NS = 0 are accepted.

A device running ARM TrustZone boots up in the secure world. After the secure world finished its initial setup by booting the TOS, it switches to the normal world and boots the Legacy OS (LOS). Most TrustZone-enabled devices are configured to use *secure boot* (cf. Section 2.2.1), i. e., the boot loader cryptographically checks the TOS prior to execution [24]. For example, the device vendor could sign the code with its private key, and the vendor's code in the boot Read-Only Memory (ROM) would verify this signature using the vendor's public key. ARM suggests using On-SoC One-Time-Programmable (OTP) hardware, such as poly-silicon fuses, allowing the device manufacturer to store a unique vendor public key in each SoC [25].[8] These checks ensure that the integrity of the boot-time code in the secure world has not been compromised, e. g., by modifying binary code or data in persistent storage. In fact, many vendors lock their devices via secure boot to ensure integrity of the secure world and to prevent end-user modification. This allows them to make the secure world-part of their Trusted Computing Base (TCB).

Limitations of TrustZone. Despite TrustZone's wide-spread deployment more than a decade after TrustZone was initially released [137], a flourishing landscape of secure mobile services is largely missing. This is mainly due to the fact that TrustZone separates the platform into only *two* security domains. The normal world is occupied by legacy code, i. e., the LOS such as Android and user applications. *All* sensitive code has to share the secure world, divided into TAs managed by the TOS and isolated via virtual memory. Past incidents have shown severe weaknesses of this concept of TrustZone [329, 46, 350, 47, 365, 351, 48]. Therefore, device vendors typically limit the access to the secure world for third-party developers, i. e., they use the secure world exclusively for their own purposes.

Google's ProjectZero [49] summarized the core limitations of TrustZone's design: (i) TrustZone's weak isolation between TAs inside the secure world leading to (ii) TCB expansion, and (iii) extensive platform access privileges of the secure world, making TrustZone a valuable attack target.

Device vendors established very strict policies for software admitted to the secure world in order to prevent adversaries from gaining control over the secure world. Each TA admitted to the secure world enlarges the platform's TCB, as it can potentially attack and compromise the TOS and other TAs. Furthermore, each TA is a potential attack target enlarging the attack surface of the secure world. Hence, each TA must be trusted, requiring extensive security assessments of third-party TAs that come with large

8 In order to minimize OTP memory cost a cryptographic hash of the vendor public key can be stored.

management overheads [166], which in turn induces high monetary costs for application developers.

As a result of these limitations, TrustZone's isolated execution environment is, in general, unavailable for third-party software developers.

2.3.2 *Intel Software Guard Extensions*

Software Guard Extensions (SGX) is an extension of the x86 Instruction Set Architecture (ISA) introduced with the 6th generation of Intel's Core processors (code name Skylake) [267, 193, 20, 206]. It introduces new instructions for creating and managing isolated software components, called *enclaves*. Enclaves are isolated from all software running on the system including privileged software, thus providing a protection mechanism that is orthogonal to the traditional hierarchical model (cf. Section 2.1). In particular, enclaves are isolated from the OS and the hypervisor as well as the System Management Mode (SMM), firmware (BIOS/UEFI), and other enclaves.

Memory Isolation. On SGX enabled platforms, programs can be divided into two parts, an *untrusted part* and an isolated, *trusted part*. The trusted part, called *enclave* in SGX terminology, is located in a dedicated partition of the physical RAM, called Enclave Page Cache (EPC). Enclaves are loaded as part of a host process and are embedded in its virtual memory, similar to a library.

SGX dedicates a fixed amount of the system's main memory (Random Access Memory (RAM)) for enclaves and related metadata, called EPC. For current systems, this memory is limited to 128 MB which is used for both, SGX metadata and the memory for the enclaves themselves. As a result, enclaves can only utilize about 96 MB of physical memory. The EPC memory is reserved in the early boot phase and is static throughout the run time of the system.

Enclave memory isolation is enforced by SGX's extended memory access control for the EPC. In particular, all other software on the system, including privileged software such as the OS, hypervisor and firmware (BIOS/UEFI) cannot access enclave memory.[9] Enclaves cannot access memory in the EPC that was not explicitly allocated to them.

Trusted Computing Base. SGX assumes the Central Processing Unit (CPU) itself to be the only trustworthy hardware component of the system, i. e., enclave data is handled in plain-text only *inside* the CPU. Data is stored unencrypted in the CPU's caches and registers, however, whenever data is moved out of the CPU, e. g., into the Dynamic Random Access Memory (DRAM), it is encrypted and integrity protected. Decryption and integrity checks are performed when data from DRAM is loaded into the CPU's internal memory, i. e., caches and registers. This protects enclaves, for instance, from being attacked by malicious hardware components with Direct Memory Access (DMA) capabilities.

The software TCB of SGX comprises the code of an enclave itself. Additionally, Intel's architectural enclaves – Provisioning Enclave (PvE) and Quoting Enclave (QE) – must be trusted. They handle the platform's cryptographic keys, such as the provisioning key

9 Enclaves are also isolated from further privileged software subsystems such as the SMM.

and the attestation key. Hence, to ensure the confidentiality, integrity, and correct use of these keys, the PvE and QE must not be malicious or compromised.

Enclave Management. Enclaves and their memory are managed by the untrusted OS. It allocates memory from the EPC for enclaves, manages virtual to physical address translation for all enclave's memory (cf. Section 2.1.1) and copies the initial data and code into an enclave when created. All relevant actions of the OS during enclave creation are recorded and included in a cryptographic hash representing an enclave's integrity.

During run time, the OS can dynamically manage enclave memory by swapping and out enclave pages (cf. Section 2.1.1). However, SGX ensures integrity, confidentiality, and freshness of swapped pages as well as the correct virtual to physical memory mapping when a page is brought back.

The OS can also interrupt and resume enclaves – similar to normal processes – causing an Asynchronous Enclave Exit (AEX). To prevent information leakage, SGX handles the context saving of enclaves in hardware and erases all sensitive register content before passing control to the OS. When an enclave is resumed, again the hardware is responsible for restoring the enclave's context, preventing manipulations of an enclave's state.

The (untrusted) host process can invoke the enclave only through a well-defined interface.

Attestation. SGX supports local attestation as well as Remote Attestation (RA). With local attestation two enclaves on the same host system can verify each other while Remote Attestation enables a remote entity to verify that an enclave was initialized correctly and is running on a genuine SGX device.

During enclave creation, the initial code and data loaded into the enclave are measured. This measurement can be provided to an (external) party to prove the correct creation of an enclave. During local attestation, the authenticity of the measurement is ensured by the SGX hardware. For RA, the authenticity of the measurement, the integrity of the measurement, as well as the fact that the measurement originates from a benign enclave is ensured by a signature. Its freshness is ensured by a challenge provided by the external party [20]. This signature is created by a special enclave, called QE, using the platform attestation key. This allows a remote entity to verify enclave measurements.

Secure Channel Establishment and Provisioning. Furthermore, the RA feature allows for establishing a secure channel to an enclave, which in turn can be used for the provisioning of confidential data to an enclave.

The initial content of an enclave is loaded from unprotected memory; hence, it can be manipulated and is not kept confidential. Therefore, confidential data must be provisioned to an enclave over a secure channel *after* it has been created.

Once an enclave is created, its integrity is protected, i. e., its operations cannot be manipulated from the untrusted host. Hence, if the enclave creates a secret key *after* it was initialized, this key cannot be accessed by the untrusted host. With a public key cryptography scheme, e. g., Rivest–Shamir–Adleman (RSA), the secret key sk and the public key pk are generated inside the enclave. The randomness required to create a secure key is provided by the CPU directly to the enclave preventing privileged host

software from tampering with the process. In particular, the `rdrand` instruction can be used to retrieve randomness from the hardware's Random Number Generator (RNG).

Sending `pk` to an external entity allows it to encrypt data and send it to the enclave. However, to prevent Man-in-the-Middle (MitM) attacks, the external entity needs assurance of `pk`'s authenticity. This is achieved by binding `pk` to the cryptographic hash of the originating enclave. In particular, SGX's QE provides a signature over both, the enclave's cryptographic hash *and* `pk`.

The QE receives the enclave's `pk` together with the request for creating an attestation quote via SGX's local attestation mechanism. This ensures that the QE can assure that `pk` originates from a specific enclave.

Sealing. SGX's sealing capability enables persistent secure storage of data, such that the data is only available to correctly created instances of one specific enclave.

Once available inside an enclave, secret data can be encrypted using an enclave-specific key and written to untrusted storage, e. g., the hard disk. The sealing mechanism allows an enclave to use secret data across multiple instantiations.

2.4 Background on Side-Channel Attacks

Side-channel attacks on software exist in many different forms. In general, any kind of resource use that is influenced by the software's execution and can be observed by the adversary, can serve as a side channel, e. g., the use of electricity as well as effects thereof such as electro-magnetic emission, or the use of shared Central Processing Unit (CPU) caches. In this work *software* side channels are of most relevance, i. e., those that are observable by software programs running on the target machine. Therefore, we focus in this section on software side-channel attacks, excluding physical or hardware side-channel attacks.

In the realm of software side-channel attacks, several distinct variants exist that can be categorized according to different aspects. On one hand, side-channel attacks can be categorized based on the shared resource used to leak information, for instance the different caches of the CPU, or the virtual memory management mechanism. On the other hand, side-channel attacks can target different information, including sensitive access patterns to data as well as secret dependent code execution paths.

We focus on controlled channel and cache side-channel attacks, which are particularly relevant in the context of Software Guard Extensions (SGX) and for this work. A detailed discussion of these attacks is provided in Section 4.4.1, where we discuss their differences to our own side-channel attack [68] and elaborate on the effectiveness of our side-channel defense [72] against these attacks.

2.4.1 *Controlled-Channel Attacks*

Xu et al. [412] demonstrated page-fault side-channel attacks on SGX, where an untrusted Operating System (OS) exfiltrates secrets from enclaves by tracking memory accesses at the granularity of memory pages. Memory access traces of enclaves can be generated

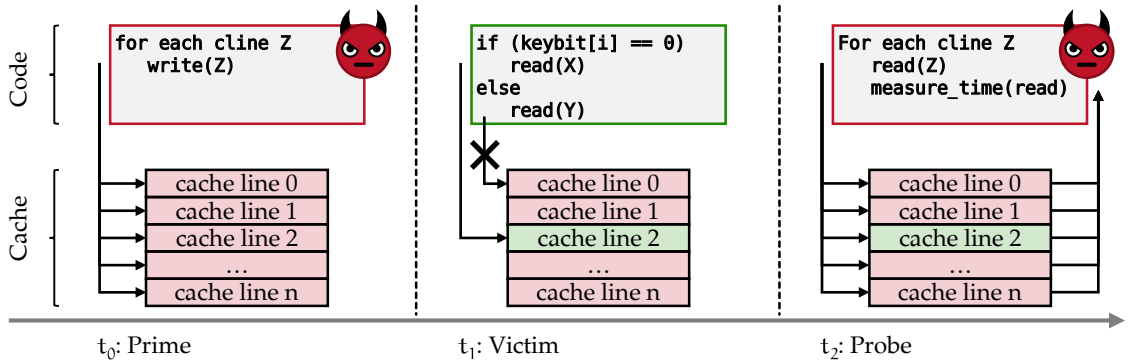


Figure 13: Prime+Probe side-channel attack technique; first the adversary primes the cache, next the victim executes and occupies some of the cache, afterwards the adversary probes to identify which cache lines have been used by the victim. This information allows the adversary to draw conclusion on secret data processed by the victim process.

using various techniques [389, 183, 176], e. g., memory segmentation [183] or Translation Look-aside Buffer (TLB) observations [176].

2.4.2 *Cache Side-Channel Attacks*

Using branch shadowing, i. e., monitoring the Branch Target Buffer (BTB), control flow of an enclave can be inferred, as demonstrated by Lee et al. [239]. Similarly, the directional branch predictor can be used to extract control flow information from enclaves, as shown by Evtyushkin et al. [143].

Cache side-channel attacks targeting SGX enclaves by monitoring data caches have been using the L1 cache [277, 174, 68, 119, 184, 405] as well as L3 cache [239].

Yarom et al. [415], Moghimi et al. [278] have investigated the possibility of leaking information through a side-channel with granularity smaller than a single cache line.

CacheBleed [415] exploits cache *bank conflicts* to leak fine-grained information. However, this attack does not apply to SGX CPUs due to an updated cache design.

MemJam [278] uses read-after-write false dependencies to introduce latency when a victim program reads data with a specific page offset. By measuring the run time of the victim program, a high number of times while *jamming* different page offsets, the adversary can infer which offsets are read more often by the victim. This attack can leak information with a four-byte granularity, however, is very noisy and requires an extremely high number of repetitions (*50 million runs*).

Exploiting the CPU's speculative execution unit Foreshadow can extract all data from SGX enclaves on non-updated platforms [390].

Prime+Probe Cache Monitoring Technique. The main steps of the Prime+Probe [294] attack are depicted in Figure 13. First, at time t_0, the adversary *primes* the cache, i. e., the adversary accesses memory such that the entire cache is filled with data of the attacker process.[10] Afterwards, at time t_1, the victim executes code with memory accesses that are

10 To prime all cache sets the adversary needs to write to #cachesets cache pages, see Section 2.1.2 for details.

dependent on the victim's sensitive data, e. g., a cryptographic key. The victim accesses different memory locations depending on the currently processed key-bit. In the example in Figure 13, the key-bit is zero, therefore address X is read. Address X is mapped to *cache line* 2, hence, the data stored at X are loaded into the cache and the data that were present in *cache line* 2 before are evicted. The data at address Y are not accessed and therefore the data in *cache line* 0 remains unchanged.

At time t_2, the adversary *probes* which of the adversary's data got evicted, i. e., which cache lines were used by the victim. A common technique to check for cache line eviction is to measure access times. The adversary reads from memory mapped to each cache line and measures the access time. If the adversary's data are still in the cache the read operation returns them fast. However, if the read operation takes longer, the data were evicted from the cache and had to be loaded from slower memory. In Figure 13, the adversary will observe an increased access time for *cache line* 2. Since the adversary knows the code and access pattern of the victim, the adversary knows that address X of the victim maps to *cache line* 2, and thus, that the sensitive key-bit must be zero. This cycle is repeated by the adversary for each sensitive key-bit that is processed by the victim until the adversary learned the entire key.

3 SECURITY ARCHITECTURES

Embedded and mobile systems are at the core of many security-sensitive and safety-critical applications, such as mobile banking or critical infrastructure. Security architectures from mobile and embedded systems provide the foundation to protect these vital use cases.

In this chapter we present security architectures for mobile and embedded platforms. In Section 3.1 we present the TyTAN security architecture for embedded real-time systems. To the best of our knowledge, TyTAN is the first security architecture for embedded systems that provides (1) hardware-assisted strong isolation of dynamically configurable tasks and (2) real-time guarantees. TyTAN's security guarantees and real-time capabilities were utilized in a collaboration project with automotive industry partners, which had the goal to establish the integrity of all Electronic Control Units (ECUs) in a car that are connected via the in-vehicles network. In Section 3.2 we present Sanctuary, a security architecture for platforms with ARM processors, which is the predominant processor-type in mobile devices. Sanctuary extends and enhances ARM's TrustZone architecture to support enclave-like Trusted Execution Environments (TEEs). Sanctuary's strong two-way isolation allows it to tolerate potential malicious applications executing in the isolated environments it provides, thus overcome the main restriction of the deployed ARM TrustZone ecosystem.

3.1 TyTAN: Tiny Trust Anchor for Tiny Devices

Already today embedded systems constitute the backbone of most safety and security critical applications. Current developments in industry, in particular the trend to "connect the unconnected" in the Internet of Everything, will carry the crucial role of embedded systems forward. The term *embedded system* is generally used to refer to a large variety of systems ranging from micro-controllers with minimal functionality to relatively powerful systems such as smartphones and enterprise routers [115]. In this section, we focus on resource-constrained embedded systems, such as Intel's Siskiyou Peak [324].

Embedded systems generate, process, and exchange vast amounts of security and safety critical data as well as privacy sensitive information, and hence are appealing targets for various attacks. Recent studies have revealed many security vulnerabilities in embedded devices [111, 115, 313, 233, 93, 274, 359, 202]. This poses new challenges on the design and implementation of secure embedded systems that typically must provide multiple functions, basic security features, and real-time guarantees at minimal cost. To ensure the correct operation of these devices, it is crucial to assure their integrity, in particular the integrity of their code and data.

Most established hardware security solutions, such as Trusted Platform Module (TPM) [383], do not scale to embedded systems due to their high complexity and costs [409, 369, 295, 267]. Software-based solutions [217, 342, 343, 244], on the other hand, typically rely on strong assumptions that are hard to achieve in practice [32].

Previous approaches that specifically target low-end embedded devices do not meet the real-time requirements of many embedded applications, or are highly inflexible, for instance, they assume a static software configuration and do not allow dynamic loading of applications at run time [139, 368, 286, 226] (see Section 3.3.2 for a more detailed discussion).

Goals and Contributions. The goal of this work is to design and develop a security architecture for small embedded devices, which allows dynamic loading of isolated tasks that might originate from mutually distrusting stakeholders. In particular, tasks must be isolated from all other tasks as well as all privileged software, e. g., the Operating System (OS).

Isolated tasks should be able to exchange information with other isolated tasks and with external entities in a secure way, i. e., providing integrity, confidentiality, freshness, and authenticity of the exchanged information.

Furthermore, the platform must be able to provide real-time execution guarantees even in the presence of potential malicious (isolated) tasks.

- We present TyTAN, which, to the best of our knowledge, is the first security architecture for low-end embedded systems providing (1) a hardware-assisted dynamic Root of Trust (RoT) that allows secure task loading at run time; (2) secure Inter-Process Communication (IPC); (3) local attestation as well as Remote Attestation (RA); and (4) real-time guarantees. TyTAN is designed for multi-stakeholder scenarios and allows for secure execution of mutually distrusting tasks.

- Our TyTAN implementation is based on Intel's Siskiyou Peak [324], an architecture intended for deeply embedded systems.
- In our evaluation we show TyTAN's efficiency and effectiveness. We show that all of TyTAN's components are real-time compliant and demonstrate its applicability to automotive embedded control systems.

3.1.1 *Requirements*

Safety and security critical applications of embedded systems, for instance, in automotive use cases, require: (1) real-time guarantees; (2) isolation of system components; (3) dynamic configuration; (4) techniques for device integrity verification (RA); and (5) support for multiple, potentially mutually mistrusting, stakeholders. All these requirements should be realized while relying on a minimal Trusted Computing Base (TCB), both in software and hardware. Additionally, the system should work with existing hardware security architectures for embedded systems.[1]

Real-time Guarantees. Acting reliably within strict time frames is highly relevant, yet not considered by most security architectures for embedded devices [139, 368, 286]. Real-time capabilities required for systems to be usable in safety critical applications.

Isolation. Faults in one system component cannot (directly) influence other components and an adversary is detained from manipulating security critical system parts or leaking secrets. Isolation is fundamental to protect critical components against unintended access by other (malicious) components.

Dynamic Configuration. Tasks (applications) can be dynamically loaded, unloaded, started, and stopped on demand at run time. Dynamic task creation enables better utilization of embedded system's resources, e. g., when a component or function is rarely used, it can be loaded on demand only when needed. This way it does not continuously allocate resources. Furthermore, dynamic task management improves the system's security as it allows updating vulnerable software components at run time. By unloading the old version of the component and dynamically loading the patched version, the system can be updated.

Integrity Verification and Secure Provisioning. While *local attestation* allows different components on the same system to mutually verify their integrity, *Remote Attestation (RA)* allows a device to prove the integrity of its software state to other devices. Furthermore, secure provisioning of confidential data to secure tasks must be enabled, e. g., by establishing an authentic and secure channel between a task and other local or remote entities.

Multiple Stakeholders. Embedded devices increasingly execute tasks from multiple, mutually distrusting stakeholders. Even smartcards compliant to the JavaCard standard can execute third party code [293]. Also in automotive Electronic Control Units (ECUs),

1 TyTAN builds on top of TrustLite [226], a security architecture for embedded system enabling isolated execution environments (cf. Section 3.3).

often software provided by the component supplier and the car manufacturer run in parallel on the same device. While the component supplier requires protection of its intellectual property and the integrity of its software components, the car manufacturer wants to ensure the correct and reliable operation of its tasks. This requires both, means to minimize the required trust between the individual software providers and stakeholders as well as mechanism to establish explicit trust between them when needed.

3.1.2 *System and Trust Model*

TyTAN targets multi-stakeholder scenarios where mutually distrusting parties provide software for real-time embedded systems. Therefore, multiple entities are involved during deployment and operations of a TyTAN system.

Stakeholders. TyTAN's model involves the following parties: the *device manufacturer* $\mathcal{M}$, the *device owner* $\mathcal{O}$, and multiple *task providers* $\mathcal{P}_i$. In the context of embedded systems applications are usually called *tasks*. $\mathcal{M}$ provides the platform's underlying hardware and software components, comprising TyTAN's TCB, which is critical for the correct operation of TyTAN (marked as *trusted software* in Figure 15). These parts must be trusted by all parties. $\mathcal{O}$ controls the OS. The OS and the tasks provided by $\mathcal{P}$ (e. g., Task A - D in Figure 15) are mutually distrusting. All tasks are isolated from each other and *secure tasks* are in addition isolated from the OS.

3.1.2.1 *Adversary Model*

The adversary $\mathcal{ADV}$'s goal is to break the security guarantees of any secure task that is not provided by $\mathcal{ADV}$ itself. $\mathcal{ADV}$ can compromise or deploy normal tasks as well as secure tasks. Furthermore, $\mathcal{ADV}$ can compromise the Real-time Operating System (RTOS) and other privileged software outside the TCB. However, if $\mathcal{ADV}$ has gained control over the RTOS, it does not aim at affecting the platform's availability. We discuss the distinctive trust layers of TyTAN below.

Hardware attacks are out of scope, hence, $\mathcal{ADV}$ cannot extract information protected by TyTAN's hardware security architecture, e. g., via side-channel attacks, or manipulate the TCB's software components, which are protected by secure boot.

Trust Relations. The device manufacturer $\mathcal{M}$ is trusted by the device owner $\mathcal{O}$ and by all task providers $\mathcal{P}_i$, shown by the green arrows in Figure 14. The device manufacturer $\mathcal{M}$ ensures that the security critical parts of the system (i. e., the TCB) are integrity protected. The TCB is protected by hardware security mechanisms, e. g., Execution-Aware Memory Protection Unit (EA-MPU)[2] and secure boot.

The device owner $\mathcal{O}$ has control of all software components on the system except for those software components that are explicitly protected, i. e., integrity protected components started with secure boot and isolated by the hardware isolation mechanism (EA-MPU). All other software components, including privileged software such as the Operating System, can be manipulated by $\mathcal{O}$. However, $\mathcal{O}$ does not trust the providers of

2 An EA-MPU can restrict access to data depending on the currently executed code (cf. Section 3.1.3.2).

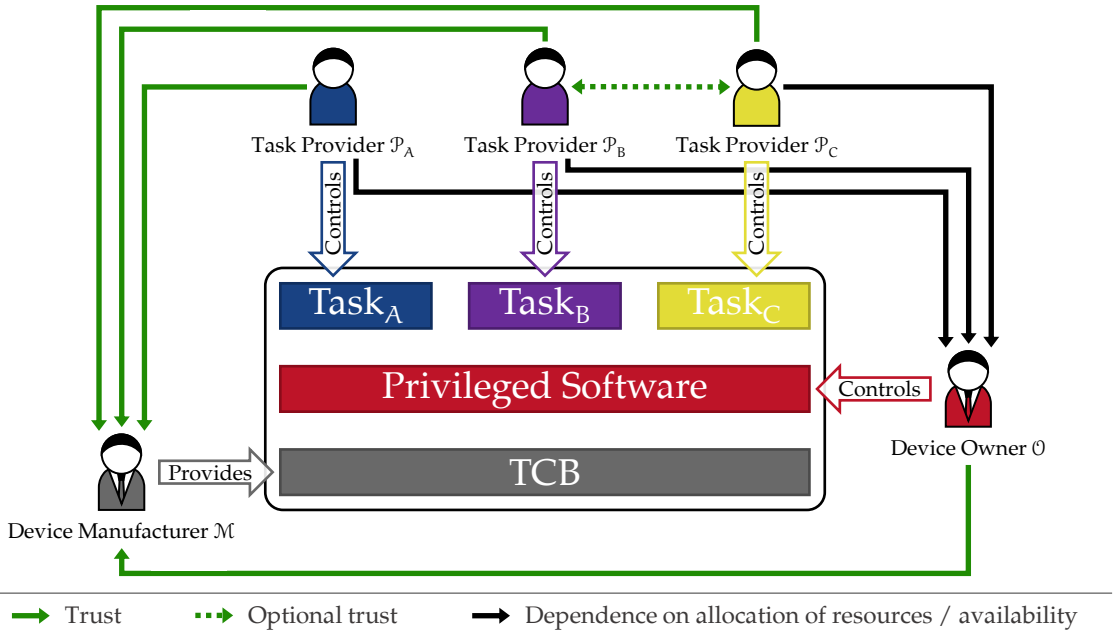


Figure 14: Trust relations between different stakeholders in TyTAN.

tasks running on the platform. Therefore, $\mathcal{O}$ isolates the privileged software (i. e., the OS) from all tasks.

Task providers $\mathcal{P}$ do not trust any part of the system except for the TCB. Furthermore, $\mathcal{P}$ are not trusted by any other party, i. e., $\mathcal{M}$, $\mathcal{O}$ and all other $\mathcal{P}$. However, all task providers have to trust and rely on the device owner with regard to resource access (indicated by the black arrows in Figure 14), i. e., $\mathcal{O}$ can simply refuse to execute a task, provide network services, etc. Furthermore, any $\mathcal{P}$ can freely decide to trust other tasks on the system and collaborate with them (green dashed arrows in Figure 14), in particular, after establishing secure channels between secure tasks where both involved tasks are authenticated.

Trust Layers. TyTAN has three different layers of trust. The first layer is the TCB (marked green in Figure 15). This layer is trusted by all parties and is protected by the hardware. The second layer is privileged software, such as the OS, and is generally untrusted, colored red in Figure 15. However, since it has control of the resource usage it can deny the execution of the system or parts of the system. The third layer comprises the tasks running on the system. "Normal" tasks are completely untrusted, they are shown as white boxes in Figure 15. They are not integrity protected; thus, they can be manipulated by the OS and depend on both, the system's TCB *and* all privileged software. Secure tasks are integrity protected and can be trusted by their respective task providers (violet and blue in Figure 15). Optionally, task providers can establish further trust relations among each other. Importantly, secure tasks as well as normal tasks cannot disturb the operation of the platform.

Trust Model for Task Providers. Figure 15 shows the trust model from the perspective of task provider $\mathcal{P}_A$, providing *Secure Task A*. Naturally, $\mathcal{P}_A$ trusts its own task. Furthermore, $\mathcal{P}_A$ has to trust the platform's TCB, marked green. $\mathcal{P}_A$ does not need to fully trust the OS, this is enabled by TyTAN's security architecture that isolates Secure Task A from the OS. However, the OS has still control of the resource usage on the platform and could refuse to execute tasks. Therefore, the OS could launch a Denial-of-Service (DoS) attack against Secure Task A (and any other task), which means $\mathcal{P}_A$ has to trust the OS with regard to availability.

$\mathcal{P}_A$ does not need to trust any other task because all tasks – secure tasks and "normal" tasks – are isolated from each other. In a multi-stakeholder system, $\mathcal{P}_A$ might not know and trust the other task providers and their tasks coexisting on the platform. However, even if $\mathcal{P}_A$ does not assume other task providers to be malicious, other tasks might be attacked and controlled by an adversary $\mathcal{ADV}$ threatening $\mathcal{P}_A$'s task without TyTAN's isolation guarantees.

Nevertheless, $\mathcal{P}_A$ can decide to have a secure and authenticated connection to a subset of the other tasks. For instance, $\mathcal{P}_A$ could collaborate with *Secure Task B*, i. e., trust $\mathcal{P}_B$. While establishing a communication channel, tasks can mutually verify their integrity and gain trust in each other.

3.1.2.2 *Hardware Platform*

We focus on resource-constrained embedded systems as used in many automotive and industrial applications. The term *embedded system* is widely used for a large variety of systems reaching from micro-controllers-based systems with minimal functionality to quite powerful systems such as home routers or smartphones. Sometimes even enterprise routers are attributed as embedded systems, even-tough they are as big, complex, and powerful as full-flashed servers [115].

Our prototype is based on a system consisting of approximately 400,000 gates. Produced in a fairly old structure size, this results in a chip in the scale of 1 mm^2. Furthermore, the system has very low power consumption (1 mW/100 Mhz). Typical examples of devices that fall into this category are the TI MSP430, ARM Cortex M3, and Intel Siskiyou Peak. All devices need to be equipped with an EA-MPU in order to be used for TyTAN. We used Intel's Siskiyou Peak with an EA-MPU [226] as base platform for TyTAN. The EA-MPU concept that was introduced in TrustLite [226] is described in detail in Section 2.1.1. For the remaining of this section we will consider this class of systems when using the term *embedded system*.

3.1.3 TyTAN *Design*

TyTAN combines different functional and security components in an architecture that provides both real-time capabilities and strong isolation for security critical tasks. TyTAN's architecture and its main components are described subsequently.

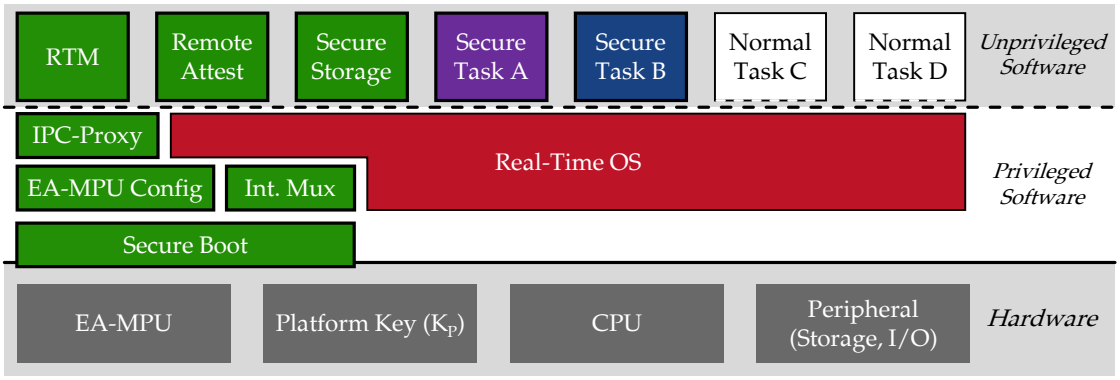


Figure 15: TyTAN system architecture providing secure tasks that are mutually isolated and isolated from the RTOS through strong hardware-based isolation, indicated by bold borders. Normal tasks are not isolated from the RTOS, shown by the dashed line. Components marked green comprise TyTAN's TCB.

3.1.3.1 TyTAN *Architecture*

TyTAN combines hardware security features with real-time capable software components. It provides TEEs that allow the isolated execution of security critical tasks without trusting the system's privileged software components, in particular the OS. Hence, TyTAN provides Trusted Execution Environments (TEEs) for embedded systems that are conceptional similar to Software Guard Extensions (SGX) enclaves. TyTAN goes beyond other platforms that aim to achieve similar isolation properties by providing real-time guarantees, which requires that all functionalities of the system have the property of either having bounded execution time[3] or being interruptible. TyTAN ensures that all security critical components are not endangered by these properties.

TyTAN adapts existing components, e. g., the RTOS, in order to integrate them with the underlying security architecture TrustLite [226]. TyTAN extends the traditional hierarchical isolation model of computer architectures and creates a new, orthogonal delimitation between trusted and untrusted system components. This allows TyTAN to utilize features and capabilities of the real-time RTOS to create a secure real-time capable system with a minimal TCB, i. e., the TCB does not need to contain complex functionalities such as a real-time scheduler.

Figure 15 shows TyTAN's architecture. At the hardware level, TyTAN relies on the security guarantees provided by an embedded system security architecture such as TrustLite [226], which provides basic mechanisms such as memory isolation, a platform key and secure boot capabilities. The hardware capabilities are managed by privileged software. TyTAN divides the privileged software into components that are critical (green in Figure 15) and those that are non-critical for security. TyTAN's goal is to minimize the amount and complexity of critical components.

Critical software components are responsible for managing the security functionalities of the system, e. g., by managing the access to the platform key or configuring memory

3 The maximal length of uninterruptible execution is application dependent. In Section 3.1.6 we evaluate the execution times of TyTAN's uninterruptible components.

isolation in the EA-MPU. All uncritical tasks are managed by the untrusted OS comprising the majority of privileged software. The critical components are protected from the untrusted components utilizing the underlay platform isolation mechanisms, i. e., using the EA-MPU. Secure boot ensures the integrity of all critical software components that are loaded at boot time. The *EA-MPU configuration module* takes control of the EA-MPU before any untrusted code is loaded, hence, it can maintain the isolation of itself and all other critical components throughout the run time of the system.

TyTAN delegates as many tasks as possible to unprivileged software. This allows TyTAN to minimize complexity and allows these components to benefit from the functionalities provided by the privileged system software. In particular, security critical tasks, such as the Root of Trust for Measurement (RTM) or the RA services, can rely on the real-time scheduling capabilities of the RTOS. TyTAN ensures these components' correct behavior even if their execution is interrupted by the RTOS, thus they do not need to have bounded execution times without violating the real-time properties of the platform.

TyTAN maintains legacy compatibility by supporting "normal" tasks that are isolated from other tasks on the system. However, their memory is not protected from accesses by the untrusted RTOS. Secure tasks, on the other hand, are protected from the RTOS, i. e., the integrity and confidentiality of their memory is ensured by TyTAN. Nevertheless, the RTOS can dynamically manage both "normal" and secure tasks, i. e., create, destroy and suspend them during run time.

For secure tasks TyTAN provides a set of security functionalities, in particular, all secure tasks are measured when created. This measurement serves as an identity for each tasks and can be used for remotely attesting the integrity of a task, locally identifying tasks to establish secure channels between tasks, and to store data securely allowing secure tasks to maintain data confidentiality across multiple executions.

3.1.3.2 TyTAN *Components*

TyTAN, see Figure 15, is composed of several components that are described in the following. The hardware and components shown in green comprise TyTAN's TCB. The Operating System (OS) is not part of the TCB.

Tasks. TyTAN supports two types of tasks: *normal tasks* are accessible by the OS, they are only isolated from other tasks. *Secure tasks* are isolated from all other software including the OS. Each secure task t has a unique identifier id_t, i. e., the hash digest of its binary code, which is measured during task creation. All tasks are loadable, unloadable, and suspendable at run time.

TyTAN ensures that secure tasks are isolated throughout their entire lifetime guaranteeing their integrity and confidentiality during execution. Furthermore, TyTAN ensures the deletion of all information from memory before accesses is given to untrusted components, e. g., when a task is unloaded, the task's memory is erased before handed back under the control of the OS. Similar, registers are erased before execution is passed to the OS, e. g., when a secure task is interrupted or exits. The initial integrity of secure tasks is measured by TyTAN and can be validated via remote and local attestation.

Real-time Operating System (RTOS). TyTAN adapts and extends an existing RTOS (FreeRTOS [15]) and inherits the real-time properties of this OS, i. e., TyTAN provides real-time scheduling for "normal" tasks as well as for secure tasks. It ensures that all tasks and system components can be interrupted to allow other pending operations to proceed within the time frame allocated to them.

Our modifications and extensions to FreeRTOS do not violate the system's underlying assumptions for a RTOS. In particular, a real-time kernel has to fulfill the following requirements as identified by Stankovic and Rajkumar [362]:

1. Multi-tasking support
2. Priority-based preemptive scheduling
3. Bounded execution time for primitives
4. High-resolution real-time clock
5. Special alarms and time-outs
6. Real-time queuing
7. Delay processes, i. e., interrupt and resume task execution

Multi-task support, priority-based scheduling, high-resolution clock, alarms, and time-outs as well as the real-time queuing of FreeRTOS are not influenced by TyTAN. However, the ability of interrupting and resuming tasks had to be adapted with the introduction of secure tasks. We will detail on TyTAN's handling of secure task in Section 3.1.4.2 and explain how secure task can be managed by FreeRTOS's preemptive scheduler. Another important property that is affected by TyTAN's extensions is requirement 3 (bounded execution time for primitives). As we already identified in our requirements analysis (cf. Section 3.1.1), TyTAN is designed such that all primitives have a maximum execution time or to be interruptible. In Section 3.1.6 we evaluated the execution time of all primitives we added to FreeRTOS and show that all uninterruptible executions have bounded execution times.

The OS is not included in TyTAN's TCB, i. e., it is not trusted with respect to the security properties of secure components and tasks of the platform. Therefore, all operations by which a malicious OS could violate these properties, e. g., modify a secure tasks initial memory content, are validated by TyTAN. As the OS is in control of the systems resources and controls, for instance, the execution schedule of tasks and access to peripherals, it can prevent the loading or execution of tasks. However, the OS can only impair availability of the platform or individual tasks.

Central Processing Unit (CPU) and Peripherals. The TyTAN platform comprises universal hardware components including processing units, i. e., a CPU, and peripheral devices, such as network interfaces, storage devices, etc. These components are part of TyTAN's TCB.

Peripherals are accessed through memory-mapped configuration registers. This allows controlling access to devices via memory access control mechanisms, i. e., using the

EA-MPU access to peripherals can be provided (exclusively) to secure tasks. The CPU is shared, executing trusted and untrusted software. TyTAN ensures that no information is leaked between consecutively executed entities by erasing the CPU's internal state, e. g., registers, before executing untrusted software.[4]

EA-MPU. TyTAN utilizes an *Execution-Aware Memory Protection Unit (EA-MPU)* [226] as memory isolation mechanism. An EA-MPU is a hardware component providing: (1) memory access control enforcement based on the code that attempts to access a data region, e. g., the stack of a task can be accessed only by the task itself; (2) each task can be invoked only at a dedicated entry point; and (3) interrupts are handled such that the memory isolation rules of the EA-MPU are enforced, i. e., a (malicious) interrupt handler cannot gain any information about the state of an interrupted task. Section 2.1.1 provides a more detailed explanation of the EA-MPU.

Platform Key. The TyTAN hardware platform comes with a *platform key* K_p. Access to this key is controlled by the EA-MPU. Only trusted software components have access to K_p. Additional keys can be derived from K_p, e. g., for RA or for secure storage.

An asymmetric key pair for RA can be derived from the device's symmetric platform key K_p using it as seed to instantiate a Deterministic Random Bit Generator (DRBG) [40], which is used to in the generate of the asymmetric key pair [280]. The public key needs to be transmitted to the Certification Authority (CA) in an authentic way, e. g., it can be extracted during manufacturing the device. The CA generates a certificate for the device's public key, which is later made available to the remote verifier.

Secure Boot. TyTAN's trusted software components (i. e., EA-MPU driver, Int Mux, IPC Proxy, RTM task, Remote Attest and Secure Storage) are loaded with secure boot and isolated from the rest of the system by the EA-MPU to ensure their integrity. These components are not higher privileged than the OS and only some of them have the same privileges as the OS (e. g., EA-MPU driver).

3.1.3.3 *Trusted Execution*

TyTAN provides trusted execution by isolating secure tasks and trusted software components based on the access control enforced by the EA-MPU. Each security primitive isolated from all other system components. The individual security primitives are described in the following.

EA-MPU Driver. The dynamic handling of tasks requires the EA-MPU to be dynamically configurable. This is performed by the EA-MPU driver, which sets the memory access control rules in the EA-MPU when loading or unloading secure tasks. The EA-MPU rules for the static components (including the EA-MPU driver itself) are set during secure boot.

Attestation. In order to prove the integrity of a task t to a local or remote verifier, the *Root of Trust for Measurement (RTM)* task computes a cryptographic hash function over the binary code of each secure task when it is created. This hash digest serves as identity of the task (id_t). To meet real-time requirements, the RTM task must be interruptible during

4 TyTAN is targeted to small embedded systems that neither provide multi-core CPUs nor caches.

the hash calculation. By isolating t's memory and preventing its execution, TyTAN ensures that t is immutable while the RTM task computes id_t. This guarantees the correctness of the measurement and enables reliable verification of id_t.

The authenticity of id_t and its origin is crucial. TyTAN supports different authentication methods for local and Remote Attestation. The EA-MPU ensures that only the RTM task can modify id_t. For local attestation, id_t can be used as both identifier and attestation report of t. TyTAN' RA mechanism uses Message Authentication Code (MAC) with an attestation key K_a to prove the authenticity of id_t to a remote verifier. K_a is derived from K_p and only accessible to the *Remote Attest* task.[5]

Secure Inter-Process Communication (IPC). TyTAN enables secure communication between tasks via an IPC proxy, which forwards messages m from a sender $\mathcal{S}$ to the receiver $\mathcal{R}$. $\mathcal{S}$ copies m and the identity $id_{\mathcal{R}}$ of $\mathcal{R}$ in general-purpose CPU-registers and invokes the IPC proxy via a software interrupt.[6] The IPC proxy determines $\mathcal{R}$'s memory location and writes m and $id_{\mathcal{S}}$ to $\mathcal{R}$'s memory. This implicitly authenticates m and $id_{\mathcal{S}}$ since the EA-MPU ensures that only the IPC proxy can write to $\mathcal{R}$'s memory. To efficiently transfer large amount of data between tasks, the IPC proxy can set up shared memory that is accessible only to the communicating tasks.

Secure Storage. Secure storage is realized as a secure task. For each task, a *task key* $K_t = \text{HMAC}(id_t|K_p)$ is generated, which is bound to the task identity (id_t) and the platform (K_p). Tasks interact with the secure storage task over secure IPC, which allows the identification of the requesting task. All data a task t sends to the secure storage task get encrypted with K_t. Since id_t is included in K_t a task that tries to access data stored before will only succeed if it has the same id_t as the task that stored the data, i. e., if it is the same task.

3.1.4 *Implementation*

In this section we describe our implementation of TyTAN. First, we describe the hardware platform of TyTAN. Afterwards we explain our extensions of the OS and its functionalities. Lastly, we elaborate on TyTAN's platform security services. The source code of TyTAN was developed on a non-public Intel platform and is not available as open source.

3.1.4.1 *Hardware platform*

We implemented TyTAN on the Intel Siskiyou Peak architecture [324], a low-power, 32-bit core intended for embedded applications. Siskiyou Peak uses a flat, physical addressing model and interacts with peripherals using Memory-Mapped Input/Output (MMIO). Siskiyou Peak was extended with an EA-MPU as presented in TrustLite [226]. An EA-MPU defines access control rules to memory depending on the currently executed code (cf. Section 2.1.1). The concept of an EA-MPU is not limited to the Siskiyou Peak

5 In Sancus [286] a key derivation scheme is shown, which allows the creation of individual attestation keys per task provider $\mathcal{P}$.

6 Provisioning $\mathcal{S}$ with $id_{\mathcal{R}}$ is left to the task developer.

platform and could be implemented on other embedded processors as well. Hence, the TyTAN architecture is not limited to one specific platform and can be adapted to other processors.

We extend the static EA-MPU usages presented in TrustLite [226] with dynamic configuration of memory access control rules. We implemented TyTAN on a Xilinx Spartan-6 Field Programmable Gate Array (FPGA) running at 48 MHz.

3.1.4.2 *Operating System*

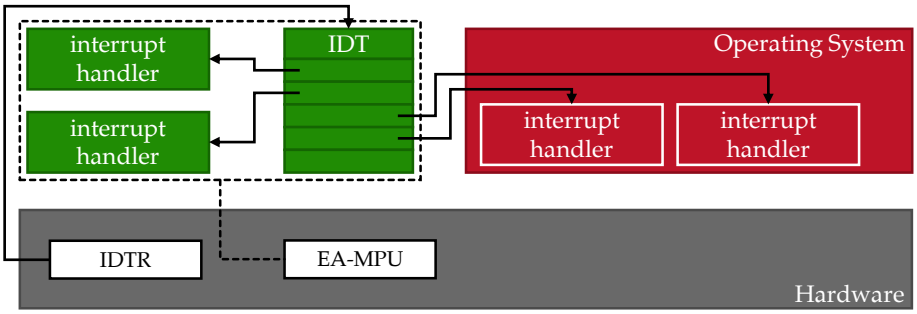
TyTAN uses the FreeRTOS [15] Real-time Operating System. We ported FreeRTOS to Siskiyou Peak and extended FreeRTOS with dynamic handling of secure tasks and support for the EA-MPU. Additionally, security primitives, such as secure IPC and secure interrupt handling were added to the OS. All our extensions to FreeRTOS were designed such that they do not violate the real-time requirements identified by Stankovic and Rajkumar [362]: (1) multi-tasking support, (2) priority-based pre-emptive scheduling, (3) bounded execution time for primitives, (4) high-resolution real-time clock, (5) special alarms and time-outs, (6) real-time queuing, and (7) delaying of processes (interrupt/resume task execution) [362].

Specifically, we made the following adaption and extensions to FreeRTOS. (i) Functionalities for dynamic task handling had to be extended, i. e., loading of tasks at run time. (ii) We extended FreeRTOS's preemptive scheduler to support secure tasks, i. e., in TyTAN the untrusted OS is not responsible to save the state of secure tasks on the task's stack nor to restore the secure task's state before resuming its execution. (iii) The OS had to become aware of Siskiyou Peak's extended hardware features (memory isolation via EA-MPU). This involves maintaining the Memory Protection Unit (MPU) rules, such as setting new protection rules on creation of a secure task. (iv) For secure IPC, a new system call interface has been added to FreeRTOS.

Interrupts. Interrupts are handled by software routines, called interrupt handlers, which are typically part of the OS. In order to invoke the correct interrupt handler for each type of interrupt the Interrupt Descriptor Table (IDT) assigns interrupt handler starting addresses to interrupt types. The hardware interrupt engine automatically invokes the corresponding interrupt handler routine on occurrence of an interrupt, i. e., the IDT determines the platform's behavior on interrupts.

TyTAN ensures the correct use of interrupt handlers by protecting the integrity of the IDT using the EA-MPU. This is crucial as the IPC proxy, for instance, should always be executed when a defined software interrupt is issued. If an adversary $\mathcal{ADV}$ manages to modify which interrupt handler is invoked by a task, e. g., to send a secure message, $\mathcal{ADV}$ could acquire confidential data or violate a message's integrity. Hence, the IDT entries (at least those of critical handlers) have to be protected from unauthorized modification. To restrict access to the IDT to TyTAN's TCB only, an EA-MPU rule protects the memory area containing the IDT, as depicted in Figure 16. The dashed line shows that the green components are integrity protected by the EA-MPU. In contrast to more powerful platform such as Intel's x86 or ARM systems the IDT in TyTAN's hardware the IDT is

set at a fixed location. Hence, the register pointing to the IDT (known as `IDTR`) cannot be modified to install a new, possibly malicious, IDT.



IDT: Interrupt Descriptor Table IDTR: Interrupt Descriptor Table Register

Figure 16: Interrupt handler protection in TyTAN. The IDT as well as handler routines for critical interrupts are protected by the EA-MPU. Uncritical interrupt handlers remain in the untrusted OS.

Dynamic Task Handling. By default, FreeRTOS does not support dynamic loading of tasks, i. e., tasks have to be loaded into memory at boot time together with the OS. However, dynamic task loading allows for better resource utilization making it a desirable feature.

Dynamic Task Loading. FreeRTOS had to be extended to allow loading tasks at run time, which requires: (1) allocation of memory for the new task; (2) loading the new task's code and static data into memory and preparing its stack: FreeRTOS operates on physical memory and the base address of a task changes depending on which memory regions are free at load time, making relocation necessary; and (3) invocation of the task, i. e., adding it to the OS scheduler.

To perform those steps, we extended FreeRTOS with an ELF (Executable Linking Format) loader. Executable Linking Format (ELF) is one of the most commonly used formats for executables; it is used, for instance, on Linux. Siskiyou Peak and FreeRTOS operate on physical memory, task binaries have to be relocatable. When loaded at run time, the base address of a task will change depending on which memory regions are currently free. ELF supports relocatable binaries and encodes all information in the binary file's header section. This includes the size of the code and data section as well as the address of the initial starting address, i. e., at which point the task should start executing. Furthermore, the ELF header contains a list of all absolute references within a binary. These references are memory addresses used by the code and depend on the memory address at which a task is loaded, i. e., they have a fixed offset from the start of a task's memory area (more precisely from the section start) and their final address is therefore the start address of their section plus the provided offset. Hence, this calculation has to be done for every reference in the relocation process after the memory location of a task is determined. However, the positions of the references themselves within the program are again expressed as an offset from the section beginning. To find a reference in memory its offset again has to be added to the sections base address. The offsets to the

references are listed in the header along with the offset to their target addresses, which have to be placed in the code.

Dynamic Task Unloading and Suspending. Unloading a task requires deleting it from the OS scheduler and reclaiming its memory. Reclaiming the task's memory by the untrusted OS requires to deactivate the EA-MPU rules isolation the memory from being accessed by the OS. However, all remaining memory content – that might contain sensitive information – has to be erased before lifting the access control policies for a secure task's memory.

Task suspending requires the OS scheduler to maintain a list of tasks that are loaded in memory without being executed at the moment. The tasks memory remains unchanged and is kept inaccessible from the OS.

Secure Tasks. Secure tasks are isolated from other software components, i. e., the memory of a secure task can be accessed only by the task itself and trusted system components. The OS is not trusted and cannot access the task's memory. Furthermore, the EA-MPU enforces that secure tasks are invoked only at a dedicated entry point to prevent code reuse attacks.

These restrictions of secure tasks effect how the OS can interact with secure tasks. We adapted FreeRTOS to reflect these restrictions when interrupting and (re)starting secure tasks.

Interrupting Secure Tasks. Tasks are frequently interrupted, e. g., to react to external events such as arriving network packets. Whenever an interrupt occurs, the hardware exception engine stops the current task and executes a predefined routine, called interrupt handler, which reacts to the interrupt and resumes the task afterwards. This entire process should be transparent to the task, i. e., after the interrupt, the interrupted task should continue execution as if it had never been interrupted,[7] which requires the interrupt handler not to change the state of the task. The task's state consists of the content of the task's memory and the CPU registers (known as the *context* of the task). While the task's memory typically remains unchanged, the CPU registers are used by the interrupt handler itself and, thus, must be saved. The *instruction pointer* (EIP) and *flags register* (EFLAGS) are saved by the hardware exception engine to the stack of the interrupted task. For normal tasks, all other CPU registers are saved to the task's stack by the interrupt handler. Since the context and stack of a secure task may contain sensitive information, the untrusted OS must not be able to access this data. TyTAN uses the trusted interrupt multiplexer (*Int Mux* in Figure 15) to securely save the context of a task to its stack before control is passed to the potentially untrusted interrupt handler. Figure 17 shows the control flow for handling an interrupt during the execution of a secure task, using a trusted software component to securely store the tasks context before executing the untrusted OS's interrupt handler. Alternatively, context saving for secure tasks can be implemented as a dedicated hardware engine, reducing latency at the cost of additional hardware, as proposed by TrustLite [226].

(Re)starting Secure Tasks. When a normal task is resumed after it has been interrupted, its context is loaded from the task's stack to the CPU-registers by the OS. The registers

7 If the task requests a service from the OS via an interrupt this has of course an effect on the task, however, in this case the change is intended.

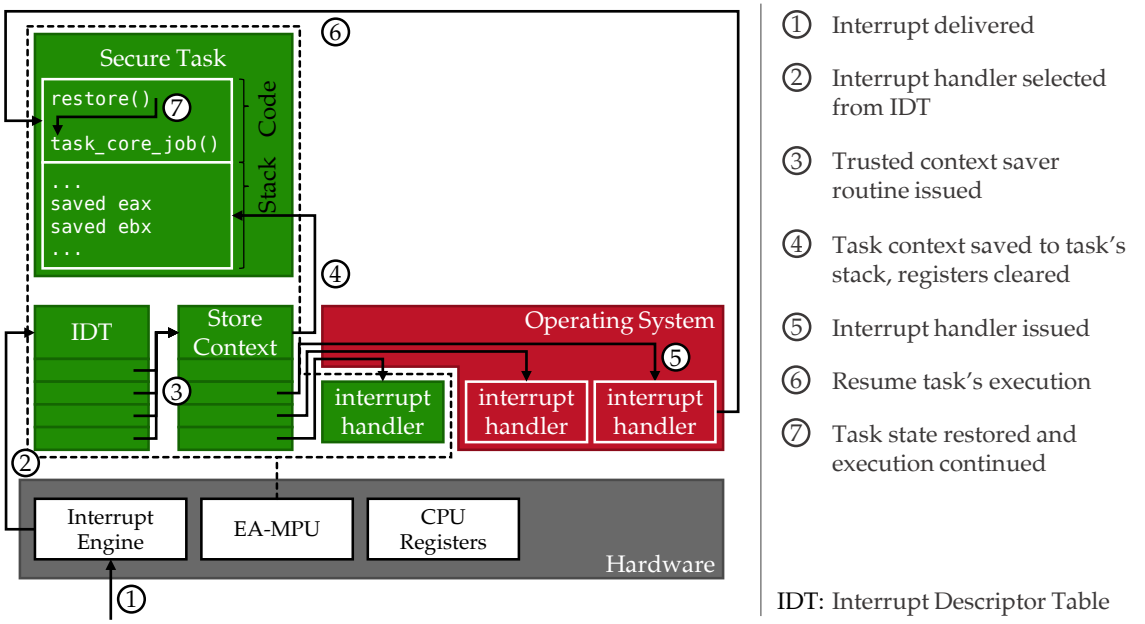


Figure 17: Secure context saving in TyTAN. Before a potential untrusted interrupt handler is executed the secure task's context, i. e., CPU register content, is saved on the task's stack. When the task is resumed it restores its own saved context before continuing, preventing the untrusted OS from learning the secure task's internal state.

saved by the hardware (EIP and EFLAGS) are restored by a dedicated CPU instruction (`iret`), which continues to execute the task from the instruction pointed to by the EIP, i. e., where the task was interrupted. When a new task is created, the OS prepares the stack of the new task as if it had been executed before and was interrupted at the starting instruction of the task, typically the `main` function. Then the OS loads the initial context of the task to the CPU registers and "resumes" the task.

For secure tasks there are two restrictions: (1) the OS cannot access the stack of a secure task to restore its context; and (2) secure tasks can be invoked only with a dedicated entry routine. This entry routine detects whether the task has been (re)started or was invoked to receive a message. TyTAN provides this information in a CPU register, which is checked by the task's entry routine. When the task has been (re)started, the entry routine restores the task's context and continues the task's execution. If the task is invoked in order to receive a message via IPC, the message receiver handler of the task is called. Since the entry routine is similar for all secure tasks, it is automatically included by the TyTAN tool chain and is not need to be implemented by the task developer.

3.1.4.3 *Platform Security Primitives*

RTM Task. When a task t is loaded, the RTM task computes the hash digest of the task t's code, static data, and initial stack layout.[8] This measurement is the basis for TyTAN's

8 In our prototype implementation we used Secure Hash Algorithm 1 (SHA1), however, the hash algorithm can be changed easily when required.

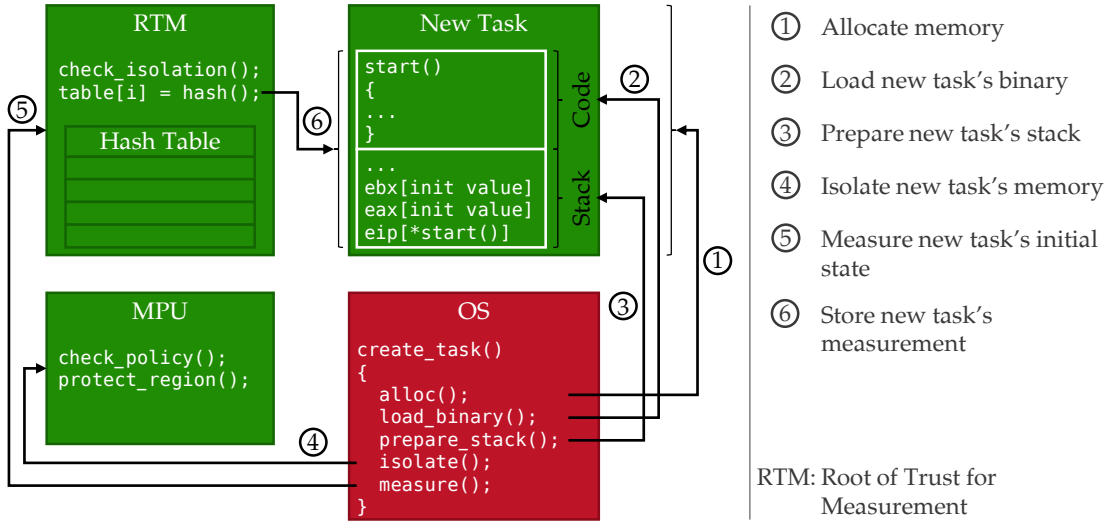


Figure 18: Creation of a new secure task in TyTAN. The new task is loaded and initialized by the untrusted OS. Before the task is executed it is isolated and its initial state is measured and recorded.

local and Remote Attestation. The integrity of the RTM task is protected by secure boot and the EA-MPU. As described before, during the loading of *t* it is subject to relocation. A measurement of a "relocated" task would only be verifiable with additional information, e. g., the memory location at which the task is loaded. To provide a position-independent measurement for tasks, the RTM task temporarily reverts the changes made during relocation before computing the hash digest.

The process of loading a new task on TyTAN is shown in Figure 18. (1) The OS allocates memory for the new task. (2) The task's code and static data is loaded by the OS similar to a "normal" task, which includes the relocation process. Next, (3) the stack of the new tasks is prepared by the OS. Any of these steps might be done incorrectly by the untrusted OS, therefore TyTAN ensures that all modifications will be reflected by the measurement of the task. (4) The memory of the new task is protected by the EA-MPU, thus the OS cannot change the task's memory any further. From this point on only trusted components (the task is not able to execute yet) can access the task's memory. (5) The memory of the new task is measured by the RTM task. Since the new task is not executable at this time and the integrity of the task's memory is protected by the EA-MPU the measurement can be interrupted without jeopardizing the correctness of the measurement. (6) The measurement is recorded by the RTM tasks, the task is marked executable and the OS gets notified that it may schedule the task.

The measurement component is implemented as a secure task that is scheduled by the untrusted OS. This means that the OS can interrupt the RTM task to make sure other tasks can meet their real-time deadlines. However, unlike other secure tasks, the RTM task is loaded in the platform's early boot phase – before the RTOS – protected via secure boot, i. e., its integrity is guaranteed.

Remote Attestation. TyTAN's RA task (*Remote Attest* in Figure 15) can attest the correct creation of a secure task to an external entity. For this the measurement of the secure task is digitally signed with a platform specific attestation key that is derived from the platform key K_p. In particular, secure tasks can request an attestation report from Remote Attest by sending a message via secure IPC. The secure tasks can include a payload in the message to Remote Attest that will be included in the signature binding the payload to the identity of the secure task. This enables the establishment of a secure channel between the secure task and an external entity.

When Remote Attest receives an attestation request from a secure task, the secure IPC mechanism provides the identity of the requesting task. Remote Attest uses the attestation key, to which it has exclusive access, to create a digital signature over the sender's identity (i. e., cryptographic hash over the task's initial memory content) and the provided payload.[9] The resulting signature is send back to the secure tasks, again via secure IPC.

Remote Attest executes as a secure task, i. e., the OS can schedule and interrupt it to ensure the real-time behavior of the system. Its integrity is guaranteed by secure boot and TyTAN's run time isolation via the EA-MPU.

Secure Storage. TyTAN realizes secure storage as a security service running as a secure task. It provides each secure task with an individual key. These individual keys can be used by secure tasks to encrypt data for storage. As these keys are derived from the platform key and the secure tasks' identities, each secure task will always have the same key, whether it was temporarily unloaded or when the platform was restarted.

A secure task can request a secure storage key via secure IPC. The secure storage task uses the platform key, which is only accessible to TCB of TyTAN, and the sender's identity, which is provided by secure IPC along with the request, to derive a task-specific key.[10] The derived key is sent back to the requesting task, again via secure IPC.

Secure IPC ensures that a secure task can only request its own key and ensures that the task's key can only be received by the requesting task. Furthermore, the requesting task gets assurance that the received key is authentic, i. e., that it was generated by the secure storage task.

The secure storage task executes as a secure task, i. e., the OS can schedule and interrupt it to ensure the real-time behavior of the system. Its integrity is guaranteed by secure boot and TyTAN's run time isolation via the EA-MPU.

Secure IPC. TyTAN's secure IPC mechanism provides (1) confidentiality of the transmitted data, (2) integrity of the data, (3) freshness and (4) authenticity for both sender and receiver. Integrity and confidentiality are ensured because only the sender and receiver task as well as the IPC proxy, which is part of the TCB, can access the sent messages. Freshness is guaranteed by the IPC proxy which will deliver each message only once to the receiver. Authenticity again is managed by the IPC proxy that will deliver messages only to the intended receiver and provides the receiver with the identity of the sender.

9 The data to be signed are compressed to a fixed length bit sting using a cryptographic hash function before calculating the digital signature.

10 The derivation can be extended to include a sender provided value allowing a secure task to request multiple different keys.

Secure IPC in TyTAN starts with the sender $\mathcal{S}$ loading a message m and the identity $id_{\mathcal{R}}$ of the receiver $\mathcal{R}$ (i. e., the measurement[11] of $\mathcal{R}$) into the CPU registers. $\mathcal{S}$ then issues an interrupt, which invokes the IPC proxy. The IPC proxy obtains the origin of the interrupt from the hardware, i. e., the hardware provides the memory address of the instruction where the interrupt was issued. From this memory address IPC proxy determines $\mathcal{S}$'s identity $id_{\mathcal{S}}$, as the code location of the interrupt issuing instruction uniquely identifies the sender task. The memory location of $\mathcal{R}$ is stored by the RTM task, which maintains a list of the identities of all loaded tasks and their memory addresses (entry points). Then the IPC proxy writes m and $id_{\mathcal{S}}$ to the memory of $\mathcal{R}$. For synchronous communication, the IPC proxy branches to $\mathcal{R}$, to enable its entry routine to process m. For asynchronous communication, the IPC proxy continues executing $\mathcal{S}$ and $\mathcal{R}$ processes m the next time it is scheduled.

3.1.5 *Security Analysis*

The primary goal of TyTAN is to assure the integrity of critical software components and secure tasks, as well as the confidentiality of data processed by these components. This is achieved through secure boot and hardware-enforced memory access control. In particular, the integrity must be ensured at load time and then maintained at run time.

Integrity. For the TyTAN components the load time integrity is guaranteed by secure boot, i. e., the platform can only boot if the software is untampered. Therefore, the TCB components of TyTAN start executing with their integrity guaranteed while no other untrusted software is executing that could violate their integrity. Before any other software is loaded and executed TyTAN's security critical components are protected by the EA-MPU. From this point on the integrity of these component is maintained throughout the entire run time of the system due to their isolation.

Similarly, tasks' memory is isolated by the EA-MPU during their run time, ensuring their integrity. A task's initial (load time) integrity can be validated by external entities before providing sensitive information to a task or accepting data from it, i. e., a secure task's initial integrity guarantees are established on demand.

Confidentiality. Confidentiality of the memory content of secure tasks and TyTAN's critical components is, similar to their integrity, ensured by the memory isolation via EA-MPU at run time. Secure data can be provisioned to secure tasks via a secure channel that can be established via RA. Using the secure storage feature, data can be kept confidential across multiple instantiations of secure tasks.

The confidentiality of platform secrets, in particular the platform key K_p, is ensured by secure boot. Only legitimate software can be loaded on the platform initially. The legitimate software will setup the memory protection for the platform secrets before any untrusted software is loaded, i. e., the platform secrets are never accessible by untrusted software ensuring their confidentiality.

11 For enhanced performance, our prototype implementation uses only the first 64 bits of the hash digest. For operational systems, the full digest of a secure hash function should be used.

Availability. Another important property of TyTAN is real-time execution of tasks, which relies on the availability of the platform. There are different attack vectors that an external adversary $\mathcal{A}dv_{ext}$ can leverage to undermine the availability of TyTAN: DoS and compromising the platform's software in order to disturb the operation of the system.

DoS attacks are domain specific (e. g., network flooding if a network interface exists, or disconnection the power supply if the device is physically accessible to the adversary), and no general solution exists to prevent DoS attacks.

To disturb the system's operations via software, an adversary $\mathcal{ADV}$ needs to gain control over the OS or a trusted software component, e. g., the EA-MPU driver. Normal tasks as well as secure task cannot disturb the operations of other components of TyTAN, due to the fact that they are isolated, and bound in their use of system recourse (e. g., execution time or memory). Hence, an adversary $\mathcal{ADV}$ controlling a task cannot disturb the availability of the platform, regardless whether $\mathcal{ADV}$ is a task provider ($\mathcal{P}$) how deployed a malicious task, or if the $\mathcal{ADV}$ compromised an initially benign task. Only if $\mathcal{ADV}$ can exploit a vulnerability in the OS, to gain higher privileges, $\mathcal{ADV}$ can affect the system's availability.

An adversary $\mathcal{ADV}$ cannot exploit TyTAN's security services to launch a DoS attack, which would prevent the system from executing its real-time tasks. In particular, TyTAN's attestation service is executed as a secure task, which can be interrupted by the RTOS. Regardless of the number and frequency of attestation request sent by $\mathcal{ADV}$, the attestation task is only executed if the system's processing time is not allocated to other tasks. Hence, $\mathcal{ADV}$ cannot occupy the system's processing resources with attestation requests. Additionally, methods presented in Section 6.1 can be used to prevent unauthorized RA invocations.

Local attestation can only be launched by an adversary controlling a secure task. The $\mathcal{ADV}$'s tasks must be scheduled for execution to issue local attestation requests exhausting the execution time slices allocated to it. The RTOS is responsible for allocating execution time slice to tasks such that they do not conflict with the real-time execution requirements of other tasks.

3.1.6 *Evaluation*

We evaluate the performance and applicability of TyTAN for embedded control systems in automotive environments. To validate TyTAN's real-time properties, we evaluate the performance of all components added or modified in TyTAN.

3.1.6.1 *Evaluation Hardware Platform*

We evaluated TyTAN on an FPGA instantiation of the Intel Siskiyou Peak processor extended with an EA-MPU. We used a Xilinx Spartan-6 FPGA running at 48 MHz with 147 000 logic cells.

Figure 19 shows our Spartan-6 FPGA mounted on a connector-board with a microphone sensor connected (on the right).

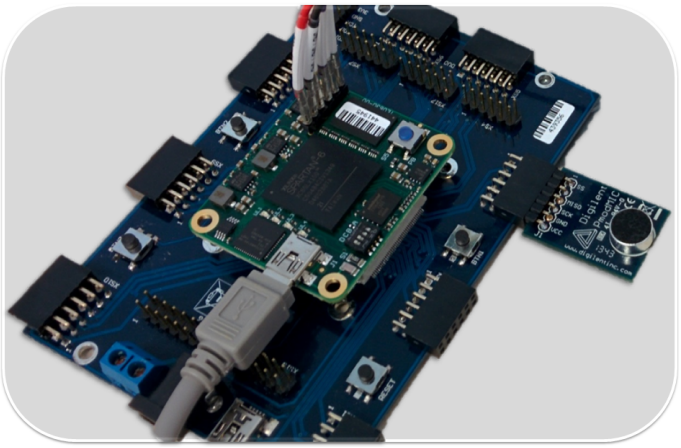


Figure 19: TyTAN prototype platform: Xilinx Sparta-6 with extension board and an attached sensor.

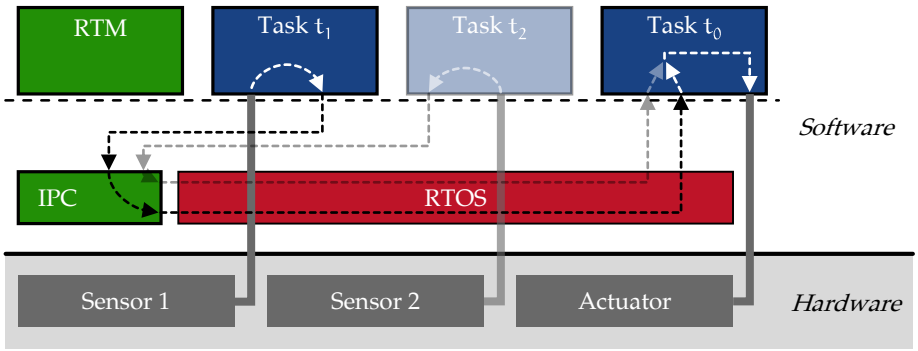


Figure 20: TyTAN use-case scenario of an adaptive cruise control system.

3.1.6.2 *Use-case Evaluation*

Our use case concerns a simulated adaptive cruise control system, where an embedded device controls the speed of a vehicle depending on the accelerator pedal position and the speed of a vehicle in front (measured by a radar sensor). The device runs three secure tasks (see Figure 20), which are scheduled on a regularly at a frequency of 1500 Hz, i. e., every 0.67 ms. Task t_1 permanently monitors the accelerator pedal position sensor. Task t_2 is loaded on demand when adaptive cruise control is activated by the driver; it monitors the radar sensor. Task t_0 controls the speed of the vehicle based on the data provided by t_1 and t_2, i. e., t_0 implements the engine control software. When cruise control is activated, t_2 is loaded into memory, which involves relocation, preparing the stack, and measuring t_2. All these operations take 27.8 ms, which is longer than the time available between two scheduling cycles of t_0 and t_1. Hence, loading t_2 could block t_0 and t_1 if the loading procedure was not interruptible. Our results in Table 1 show that t_0 and t_1, despite the fact that they have to be scheduled at high frequency, meet their deadlines while t_2 is loaded. TyTAN's capability to reliably schedule t_0 and t_1 is crucial for safe and precise control of the vehicle's speed.

Table 1: Use-case evaluation results as illustrated in Figure 20

Task	t_1	t_2	t_0
Before loading t_2	1.5 kHz	—	1.5 kHz
While loading t_2	1.5 kHz	—	1.5 kHz
After loading t_2	1.5 kHz	1.5 kHz	1.5 kHz

Table 2: Performance of saving the context of a secure task (in clock cycles)

Store context	Wipe registers	Branch	Overall	Overhead
38	16	41	95	57

Table 3: Performance of restoring the context of a secure task (in clock cycles)

Branch	Restore	Overall	Overhead
106	254	384	130

Table 4: Performance of relocation for different numbers of addresses changed by the relocation process (in clock cycles)

# of addresses	Runtime (min)	Runtime (avg)
0	37	37
1	673	703
2	1,346	1,372
4	2,634	2,711

Table 5: Performance of creating a secure task (in clock cycles)

Task type	Relocation	EA-MPU	RTM	Overall	Overhead
Secure	3,692	225	433,433	642,241	437,380
Normal	3,692	225	0	208,808	3,917

3.1.6.3 *Performance of* TyTAN *Components*

We evaluated the performance of TyTAN, in particular, all components that could have an impact on its real-time behavior, namely: (1) interrupt handling; (2) secure task creation; and (3) secure IPC.

We present all results in *clock cycles*; measurements in units of time are less meaningful because they depend on the platform' clock-speed, which is variable and depends on many factors that are not related to TyTAN.

Interrupt Handling. When an interrupt occurs, a context switch from the executing task to the OS is performed. This involves three main tasks, which we evaluated individually: (1) saving the context, (2) wiping the CPU registers, and (3) branching to the routine handling the interrupt. Table 2 shows the results of the individual steps in TyTAN and the overhead TyTAN induces compared to an unmodified FreeRTOS.

When a previously interrupted secure task should be continued, its context must be restored. For this, the OS first branches execution to the entry address of the secure task. Afterwards, the task restores its own context before continuing execution where it was interrupted. Table 3 shows our evaluation results for restoring a secure task as well as the overhead compared to an unmodified FreeRTOS.

Secure Task Creation. In TyTAN the creation process of a secure task t requires three additional operations, which we evaluated individually: (1) FreeRTOS was extended with

Table 6: Performance of configuring EA-MPU depending on the position of the first free slot in the EA-MPU with 16 slots in total (in clock cycles)

Free slot position	Finding free slot	Policy check	Writing rule	Overall
1	76	824	225	1,125
2	95	824	225	1,144
16	399	824	225	1,448

Table 7: Performance of measuring a task depending on its memory size and number of memory addresses changed during relocation (in clock cycles)

Memory size	Runtime	# of addresses	Runtime
1 block	8,261	0	114
2 blocks	12,200	1	680
4 blocks	20,078	2	1,188
8 blocks	35,790	4	2,187

the capability for dynamic loading of tasks, which implies memory address relocation in the binary. The performance is dependent on the number of memory addresses that need to be processed. Only for secure tasks two further operations need to be performed in the creation process. (2) Secure tasks are isolated by setting EA-MPU rules, including the validation of relevant security policies, e. g., ensuring that protected memory regions do not overlap. And (3) every task is measured by the RTM task. This measurement is dependent on the memory size and the number of memory addresses affected by relocation.

Table 5 compares the performance of creating a secure task versus a "normal" task, however, both types of tasks are loaded dynamically. For this evaluation, a task was used, which had a memory size of 3962 Byte and 9 memory addresses that had to be processed for relocation. While the relocation as well as the measurement – performed by the RTM task – induce considerable overhead both operations are interruptible, i. e., they do not prevent the platform from performing real-time tasks.

Relocation. For all dynamically loaded tasks, whether it is a secure task or a "normal" task, relocation has to be performed. The performance of relocation depends on the number n of addresses in task t's code that need to be updated in the relocation process. Table 4 shows our evaluation results for different n. From these measurements we can conclude that the run time of the relocation process is linear in the number of the references, with the cost per reference being $\mathtt{reloc}_{\mathtt{perRef}} \approx 650\,\mathtt{clock\,cycle}$ and fix cost of $73\,\mathtt{clock\,cycle}$. Hence, the overall run time cost for relocation of a task t with n addresses can be calculated as: $\mathtt{run\,time} \approx 37\,\mathtt{clock\,cycle} + n \cdot 650\,\mathtt{clock\,cycle}$.

EA-MPU Configuration. Configuring the EA-MPU requires three steps. First, finding a free EA-MPU slot for the new access control rule; second, checking the new rule against existing EA-MPU rules (e. g., ensure that protected regions do not overlap); and, finally,

Table 8: Memory consumption of TyTAN's OS

FreeRTOS	TyTAN	Overhead
215,617 Bytes	249,943 Bytes	15.92 %

writing the rule to the free EA-MPU slot. The number of MPU regions is fixed by the hardware configuration. For our implementation with 16 available MPU regions, setting a new rule resulted in the measurements listed in Table 6. Our evaluation shows that the first step, finding a free slot, depends on the current EA-MPU utilization. However, the worst case, i. e., only the last slot is empty, has a fixed execution time, thus it does not violate the requirements for real-time execution when executed uninterruptible.

Task Measurement. The overall time required to measure a task t depends on three factors: First, the memory size of the task t to be measured. The measurement is done using a cryptographic hash, which processes the memory in blocks of fixed length. For each block, a routine has to be repeated, hence, the execution time is linear in the number of blocks being processed. The second factor is the number of memory addresses in t changed by relocation, which need to be converted in a position independent format for the measurement. Third, the number of interruptions of the RTM task while measuring task t.

Since the interruption and scheduling of the RTM task are use case specific, we provide our evaluation results excluding this factor.

Table 7 shows the performance evaluations results for measuring a task t. These results show that the run time of the measurements depends linearly on the memory size of the measured task and the number of memory address references involved in the relocation process. Calculating the SHA1 hash of a task consists of a fixed overhead $\mathtt{sha1}_{\mathtt{fix}} \approx 4,300\,\mathtt{clock\,cycle}$ and a per block cost of $\mathtt{sha1}_{\mathtt{perBlock}} \approx 3,900\,\mathtt{clock\,cycle}$. For the reverse relocation, the fix cost is $\mathtt{reloc}_{\mathtt{fix}} \approx 110\,\mathtt{clock\,cycle}$ and the cost per reference is $\mathtt{reloc}_{\mathtt{perRef}} \approx 500\,\mathtt{clock\,cycle}$. Hence, the run time (T) of measuring a task depends on the number of blocks (b) and the number of relocated addresses (a): $T = \mathtt{sha1}_{\mathtt{fix}} + b \cdot \mathtt{sha1}_{\mathtt{perBlock}} + \mathtt{reloc}_{\mathtt{fix}} + a \cdot \mathtt{reloc}_{\mathtt{perRef}} \approx 4,300\,\mathtt{clock\,cycle} + b \cdot 3,300\,\mathtt{clock\,cycle} + 110\,\mathtt{clock\,cycle} + a \cdot 500\,\mathtt{clock\,cycle}$.

Secure IPC. TyTAN's communication performance depends on the run time of the IPC proxy and the execution time of the entry routine of the receiver task. The IPC proxy, which is part of TyTAN's TCB and executes uninterruptible, takes $1,208\,\mathtt{clock\,cycles}$. Importantly, the run time is independent of the sender task, the receiver task and the message, including message size. Hence, the IPC proxy's execution time is fixed, and thus, compliant with TyTAN's real-time requirements.

The entry routine of the receiver processing the message takes $116\,\mathtt{clock\,cycles}$ to call the receiver task's function that is responsible for handling the message. The run time of the message processing is implementation specific. However, all processing performed by the receiver task is interruptible, thus not impacting the platform's real-time capabilities.

The overall performance of the secure IPC mechanism is $1,324\,\mathtt{clock\,cycles}$.

3.1.6.4 *Memory consumption*

The memory consumption of TyTAN's OS is the amount of memory used when no task is loaded, however, the RTM task is always loaded and therefore included in our memory overhead evaluation. Table 8 compares the memory consumption of TyTAN and an unmodified FreeRTOS. It shows that TyTAN incurs moderate additional memory costs of approximately 16 %.

3.1.6.5 *Trusted Computing Base*

We have evaluated the TCB of TyTAN in terms of Lines of Code (LoC). Our measurements were made with the tool `cloc` [12].

Not all components of TyTAN's TCB are required for the system to work, therefore, we provide an evaluation for the minimal set of required components, i. e., the interrupt system, the IPC proxy, the EA-MPU components and the RTM task. Table 9 lists the number of LoC for each component as well as the total number for the entire TCB.

Table 9: Lines of Code in TyTAN's TCB

Interrupt	IPC proxy	EA-MPU	RTM	TyTAN's TCB	FreeRTOS
79	133	149	73	545	7891

Our evaluation shows the advantages of TyTAN compared to traditionally embedded system designs in which the entire RTOS has to be considered trusted. TyTAN shows a reduction of the TCB by 93 % in number of lines of code compared to FreeRTOS.

3.1.7 *Conclusion*

TyTAN is the first comprehensive security architecture for low-end embedded systems that provides (1) dynamic loading and configuration of *secure* tasks, (2) secure IPC, and (3) real-time guarantees. We implemented TyTAN on the Intel Siskiyou Peak architecture and demonstrated its effectiveness and efficiency through extensive evaluation.

The TyTAN architecture enables many new applications and use cases for low-end embedded systems. In the subsequent sections we present novel solutions for collaborative autonomous systems enabled by TyTAN's platform security guarantees.

3.2 Sanctuary: ARMing TrustZone with User-space Enclaves

Mobile devices, in particular smartphones, have become the most used personal computing devices, i. e., today, the most common way of accessing internet services is through mobile device, replacing traditional devices such as desktop and laptop computer [269]. The success of mobile devices is rooted in the software ecosystems of – often cloud-connected – services that evolved around them.

However, the success of mobile devices and their increasing use for sensitive applications, including mobile banking, payments and eID services, makes them attractive targets. The large attack surfaces of today's mobile devices, due to their feature richness and complexity, pose challenging privacy and security risks.

The need for *secure mobile services* motivated ARM in 2008 to develop a security architecture for mobile devices, called TrustZone [24] (see Section 2.3.1 for details). TrustZone divides a platform into two logically isolated systems, where one, called *normal world*, hosts all non-critical code and data. The second system, called *secure world*, is reserved for security critical services, providing them with a Trusted Execution Environment (TEE).

Unfortunately, TrustZone does not provide an open and accessible TEE solution as desired by third party application developers, which would also be beneficial for the users.

Limitations of TrustZone. In TrustZone, *all* sensitive code has to share a single TEE due to TrustZone's separation into only *two* security domains. As a result, all Trusted Apps (TAs) enlarge the platform's Trusted Computing Base (TCB), compelling device vendors to establish very restrictive usage policies, and thus, largely limiting the access of TrustZone's TEE capabilities for third party developers, as described in more details in Section 2.3.1.

In an effort to make the TEE security advantages more widely available, device vendors provide limited TEE functionalities, such as key storage, to applications in the normal world through public interfaces. However, this approach does not allow developers to protect their own security-sensitive code and data, i. e., it is not sufficient to create feature-rich and secure mobile services.

Mobile Security Architectures. Security architectures targeting ARM-based systems previously proposed by academic researchers rely on virtual memory for isolation [197, 36, 145], using the same isolation mechanism proven insufficient for isolating TAs within ARM TrustZone's secure world [49]. Approaches that rely solely on *temporal isolation*, i. e., suspending the entire normal world to provide protection for TA execution, are not suitable for today's multi-core platforms [264, 371], since they effectively disable multitasking and parallel execution for the entire platform, which imposes severe usage restrictions that directly affect user experience.

Goals and Contributions. Our main goal is to enable multiple independent TEEs on ARM-based platforms without requiring hardware changes. This allows third party application developers to benefit from make use of the full potential of TEEs.

SANCTUARY is a novel security architecture for Trusted Execution Environments (TEEs) on ARM-based systems. It provides isolated execution domains, comparable to the user-space enclaves provided by Software Guard Extensions (SGX) on Intel platforms, within the normal world. SANCTUARY allows third party apps to benefit from the same strong isolation guarantees TrustZone relies on to separate the secure world from the untrusted legacy code in the normal world. This design solves the dilemma of TrustZone by enabling security-sensitive apps – called Sanctuary Apps (SAs) – to benefit from the TEE security guarantees without endangering the security of the entire platform through the addition of potential malicious or vulnerable code to the secure world.

SANCTUARY achieves SA isolation by dynamically partitioning and re-allocating system resources: CPU cores and physical memory are temporarily reserved for isolated compartments in which SAs execute without suspending the rest of the system. SANCTUARY leverages TrustZone's Address-Space Controller (TZASC) to ensure hardware-enforced, two-way isolation between SAs and all other system components. This enables an SGX-like usage of TrustZone without requiring any hardware modifications.

Our main contributions are as follows:

- We present SANCTUARY, a novel security architecture design providing enclave-like isolated, normal-world compartments for ARM-based platforms, building on existing TrustZone hardware and software components.
- In our conference publication [73] we present a Proof of Concept (PoC) implementation of SANCTUARY, using the HiKey 960 development board and the open-source software OP-TEE for TrustZone.
- We analyze and discuss the security properties of SANCTUARY considering strong adversaries with control over all normal-world software (including the Legacy OS (LOS)) as well as malicious SAs.
- We evaluate several performance aspects of SANCTUARY and demonstrate SANCTUARY's practical performance and real-world benefits in a concrete use case. Detailed evaluation results of SANCTUARY are presented in our conference publication [73].

3.2.1 *Background*

In this section we provide the background knowledge specific to SANCTUARY. General background information, e. g., on ARM TrustZone and Intel SGX are provided in Chapter 2.

TrustZone Address Space Controller. TrustZone's memory isolation between the secure world and the normal world is enforced by partitioning the physical memory address space. All memory accesses are moderated by the TrustZone Address Space Controller (TZASC), which inspects all memory transaction on the system bus (see Figure 12) and enforces the inaccessibility of secure-world memory regions from the normal world. This enforcement is done based on specific bus transaction characteristics. The first version of

the TZASC – named TZC-380 – was published in 2010 [26], and could distinguish two types of memory accesses based on a single bit, named "non-secure" (NS): *non-secure* accesses (`NS = 1`), and *secure* accesses (`NS = 0`). All memory transactions issued by a Central Processing Unit (CPU) core carry the NS-bit indicating whether the CPU core was execution in the secure world (`NS = 0`) or in the normal world (`NS = 1`) when issuing a transaction.

The TZASC's newer version (TZC-400) was introduced in 2013. It provides extended filtering capabilities for memory transactions based on additional characteristics [27]. Its new feature, called *identity-based filtering*, allows the definition of non-secure memory regions, which are accessible only by selected bus masters, i. e., devices that can access the memory. Bus masters are identified based on a so-called *bus-master ID*, which is included in every memory transaction a bus master issues. In the ARM reference design, these bus-master IDs are assigned to individual devices, such as CPU, GPU, DMA controller. The identity-based filtering was designed by ARM for their TrustZone Media Protection Architecture [28].

3.2.2 *Adversary Model and Requirements*

3.2.2.1 *Adversary Model*

SANCTUARY's adversary model is based on the same standard assumptions as TrustZone and related work [156, 36, 35, 60, 122, 197], i. e., all normal-world software, including privileged software such as the Operating System (OS) (executing in `EL1`, cf. Section 2.3.1) and the hypervisor (executing `EL2`), are under the control of the adversary. However, secure-world software as well as the monitor mode (`EL3`) are assumed to be immune against the adversary.

The adversary is assumed to be capable of performing passive physical attacks. Invasive physical attacks, such as fault injection at run time, are out of scope.

Further, SANCTUARY does not provide availability guarantees, i. e., we do not consider Denial-of-Service (DoS) attacks.

In details, SANCTUARY's assumptions are:

- All normal-world software is considered untrusted, independent of its privilege level, including applications (`EL0`), the LOS (`EL1`) and the optional hypervisor (`EL2`).
- Privilege levels (`EL2` - `EL0`) are isolated via virtual memory.
- Software protection mechanism, e. g., Execute Never (XN), Unprivileged Execute Never (UXN), Privileged Execute Never (PXN), and Privileged Access Never (PAN) are available and active.
- Secure world and normal world are isolated through physical memory partitioning enforced by the TrustZone hardware extensions [24].
- Secure-world software, including the boot loader and the firmware (monitor mode executing in `EL3`), is trusted, i. e., comprising the system's TCB.

- Individual SAs can be compromised without risking the security guarantees of other SAs or the system's TCB.

3.2.2.2 *Requirements Analysis*

To enable mutually isolated security domains on ARM TrustZone-based systems, several security requirements as well as functional requirements, listed below, need to be fulfilled. SANCTUARY fulfills all security requirements as we show in Section 3.2.5. Further, we demonstrate that SANCTUARY meets the functional requirements by our evaluation presented in Section 3.2.6 and in our conference publication [73].

1. **Code and data integrity.** The integrity of an SA's code and data must be preserved. This can be achieved by (i) isolation during SA execution in combination with (ii) attestation of the SA code and static data when loaded into the isolated compartment.

2. **Data confidentiality.** Confidentiality of data processed in an SA must be preserved. This can be achieved by (i) a secure channel for provisioning the data, (ii) spatial isolation during execution, and (iii) temporal isolation to prevent that sensitive information becomes accessible after SA execution has finished.

3. **Secure channel to secure world.** An SA needs a secure channel to utilize security services provided by the secure world without tampering by the normal world. This can be realized via *exclusive* shared memory, i. e., shared memory accessible only by the SA and the secure world, which, at the same time, is inaccessible by untrusted normal-world software.

4. **Protection from malicious SAs.** To enable unrestricted usage models for SAs, malicious SAs must be tolerated. Protecting the platform from malicious SAs can be achieved by limiting the access privileges of SAs (i. e., SAs execute at privilege level `EL0`) and preventing them from accessing normal-world memory.

5. **Hardware-enforced resource partitioning.** To ensure strict isolation, both spatial *and* temporal isolation are needed.

6. **Minimal software changes.** Leveraging existing interfaces of the secure-world OS and the normal-world OS prevents extensive modification of the software stack.

7. **Positive user experience.** Assigning a single CPU core for limited time to SA execution leads to low impact on the overall system performance for most usage scenarios on today's commonly available multi-core architectures. Latency should be kept low by minimizing the SA run-time environment.

3.2.3 SANCTUARY *Design*

The goal of the SANCTUARY architecture is to enable secure and widespread use of TEEs (e. g., through third-party developers) on ARM-based devices. SANCTUARY allows

the creation of multiple parallel isolated compartments on ARM devices in the normal world, which are strictly isolated from the LOS and Legacy Apps (LAs). The isolated compartments, which we call SANCTUARY Instances, run security-sensitive apps called Sanctuary Apps (SAs). Every SANCTUARY Instance executes only a single SA at a time. Since all SANCTUARY instances are independent and separated from each other, SAs are strongly isolated from each other. Additionally, all SANCTUARY instances are isolated from the existing TrustZone secure world.

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

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

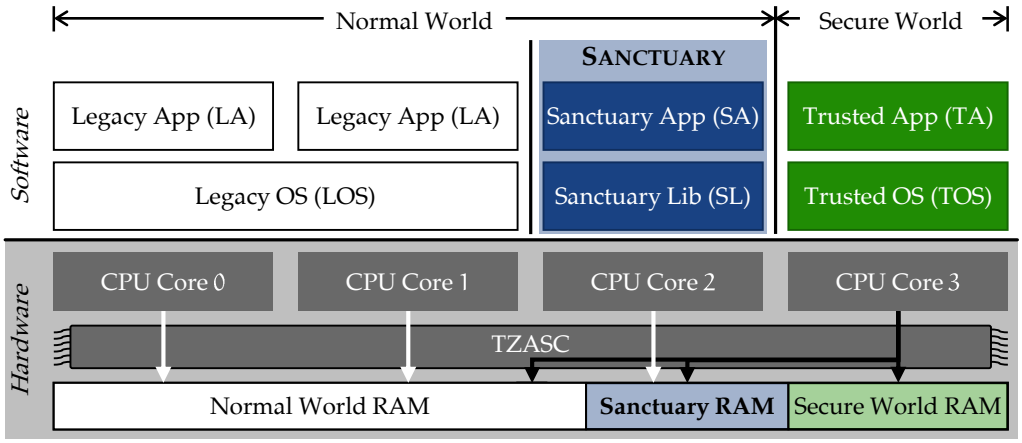


Figure 21: The SANCTUARY design reserves one CPU core in the normal world to enable isolated execution of SAs. The TCB includes the hardware (gray) and the secure-world software (green) that is involved in the initialization of an SA.

Figure 21 shows an abstract view of SANCTUARY's design. In the following, we describe SANCTUARY's isolation mechanism, its initialization, and its security services.

3.2.3.1 SANCTUARY *Isolation*

In addition to the existing security boundary between TrustZone's secure world and normal world, SANCTUARY enables isolation *within* the normal world. A dedicated memory region is made exclusively accessible by one CPU core by leveraging ARM's new memory access controller TZC-400. Details on how the controller needs to be configured to achieve this physical memory partitioning are provided in our conference publication [73]. As a result, all software executing on that CPU core is protected from

untrusted software executing on the remaining CPU cores of the system. In Figure 21, CPU core 2 running a SANCTUARY Instance is configured to have exclusive access to the *Sanctuary RAM* partition, as depicted by the arrows. The untrusted normal-world software, executing on CPU cores 0 and 1, can only access the normal-world memory. Furthermore, the CPU core assigned to the SANCTUARY Instance is *not* allowed to access normal-world memory, achieving a two-way isolation, which allows SANCTUARY to tolerate potentially malicious SAs. However, SANCTUARY does support shared memory between normal world and SA for efficient communication as well as shared memory between secure world and SA to establish a secure channel. This enables scenarios such as secure UI relying on purpose-built TAs. SANCTUARY's handling of shared memory is explained in detail in our conference publication [73].

The secure-world software is trusted and therefore allowed to access all memory, including normal-world memory, SANCTUARY memory, and secure-world memory (black arrows in Figure 21).

Multi-SA Isolation. SANCTUARY instances are either executed consecutively on the same CPU core, or execute on separate, mutually isolated CPU cores, each with a dedicated memory partition assigned. Thus, SANCTUARY supports parallel execution of multiple SAs by dedicating multiple CPU cores to SANCTUARY instances, each with its own memory partition. After SA execution finished, the system returns to its original state (see Section 3.2.3.2) and the next SANCTUARY instance can be launched. This ensures strong isolation between SAs: all SAs are executed completely independently of each other.

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

3.2.3.2 SANCTUARY *Initialization*

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

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

Next, the code to be executed on the SANCTUARY core, i. e., SL and SA, is loaded into a separate memory section. After the memory isolation has been activated, the

loaded code is validated using digital signatures. The signature for the SL is provided by the device vendor, whereas the signature for the SA is provided by the SA developer. The detailed verification process is described in our conference publication [73]. After a successful verification, the dedicated CPU core is restarted. The SANCTUARY core starts from a defined initial state, boots the SL and executes the SA.

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

3.2.3.3 SANCTUARY *Security Services*

The initial content of an SA is loaded from unprotected memory, hence, it can be manipulated and must not contain confidential data. Therefore, SANCTUARY provides a mechanism to provision confidential data to an SA over a secure channel *after* it has been created. However, to ensure that secret data is not sent to a malicious (or maliciously modified) SA, the integrity and authenticity of an SA needs to be verified before provisioning secret data. To enable secure provisioning of secret data to an SA and secure storage of secret data, SANCTUARY provides a set of security services implemented as TAs. These TAs are supplied by the device vendor; therefore, they are called vendor TAs throughout this work. These TAs run on top of the secure-world Trusted OS (TOS) (see Figure 21).

Remote Attestation (RA) allows an SA to establish a secure channel to an external entity. Through the platform identity feature of TrustZone, the integrity measurement of SANCTUARY can be authentically reported to a third party. Linking the authentic integrity report with the establishment of a secure channel to the SA, creates a secure and authenticated channel through which confidential data can be provisioned (cf. Section 2.3.2).

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

Further security services, e. g., monotonic counters, secure timers, or secure randomness, can be provided by TrustZone's secure world, as well. Similar security services are commonly available in commercial TEE implementations, for instance Intel SGX [206, 267, 193, 20] and can be implemented similarly in SANCTUARY. In addition, secure user interfaces for SAs can be easily provided by TAs, as secure I/O is already supported by TrustZone.

3.2.3.4 SANCTUARY *Software Model*

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

Additionally, by coupling each SA with an LA, the functionalities of the SL can be minimized. In particular, the LA acts as a proxy and allows the SA to make use of all functionalities provided by the LOS, such as file system access. The LA and SA can efficiently exchange information and interact with each other via shared memory. When an SA wants to provide sensitive data to the LA, e. g., for persistent storage, the SA can use the sealing service (see Section 3.2.3.3) to encrypt the data before sending it to the LA.

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

3.2.4 *Implementation*

Our SANCTUARY prototype is implemented on a HiKey 960 development board, which is equipped with a recent ARMv8 System-on-Chip (SoC). It is based on a big.LITTLE processor architecture with eight CPU cores in total, four Cortex-A73 and four Cortex-A53 cores. Importantly, the HiKey 960 board allows developers to access and modify the secure-world software.

SANCTUARY's software components adapt and extend, on one hand, OP-TEE [245], which provides an open-source implementation of a secure-world TOS. OP-TEE is extended to validate load-time integrity of SAs and to enforce run-time isolation of SAs. Furthermore, SANCTUARY's security services, e. g., for RA and sealing, are added to OP-TEE. In total, SANCTUARY adds 1313 Lines of Code (LoC) to the TCB. On the other hand, the normal-world LOS – a recent Linux distribution with support for the HiKey 960 board – is extended to dynamically manage the systems resources (CPU cores and memory) as well as SAs. The Zircon micro kernel [171] is used within each SANCTUARY Instance to provide a run-time environment (SL) to the SAs.

The details of our implementation are presented in our conference publication [73].

3.2.5 *Security Analysis*

SANCTUARY's goal is to protect against a strong adversary, as defined in Section 3.2.2.1. To achieve this goal, SANCTUARY needs to fulfill all security requirements defined in Section 3.2.2.2.

We will analyze SANCTUARY with regards to all attack vectors included in our adversary model (Section 3.2.2.1), i. e., the adversary can control (i) unprivileged normal-world execution, (ii) privileged normal-world execution, and (iii) malicious SAs. In any case, the adversary has at least one of the following two goals: (1) compromise the integrity or confidentiality of a benign SA, or (2) misuses a malicious SA to gain control over the LOS. We discuss the second case separately in Section 3.2.5.5. Subsequently we consider the first attack goal.

An attack can occur at any point in an SA's life cycle, i. e., before a SANCTUARY Instance is started, while the execution of an SA is prepared (setup), during the boot of

the SANCTUARY Instance, while an SA is executed, during tear-down of a SANCTUARY Instance, or after a SANCTUARY Instance has finished.

Before and after a SANCTUARY Instance executes, the adversary can attempt to compromise the integrity or confidentiality of stored data. In particular, the adversary can violate the integrity of an SA's binary or the SL binary (Section 3.2.5.1). Additionally, SA's data protected via SANCTUARY's secure storage mechanism can be attacked (Section 3.2.5.2).

While a SANCTUARY Instance is executing, the adversary can target SANCTUARY's isolation for code and data, either directly (Section 3.2.5.3), or via cache attacks (Section 3.2.5.4).

3.2.5.1 *Binary Integrity*

The adversary can manipulate the binaries of SL and SAs, as they are stored unencrypted in memory accessible from the normal world. To overcome with this threat, SANCTUARY measures the integrity of the binaries at load time, allowing their verification via SANCTUARY's local and Remote Attestation features. The SL binary is locally verified by the secure world during the setup of a SANCTUARY Instance. Its integrity is enforced, i. e., an integrity validation of the SL aborts the SANCTUARY Instance creation process. An SA's load time integrity can be validated by verified via Remote Attestation before provisioning sensitive information to it. Thus, compromised SAs cannot gain access to sensitive data. When an SA's binary is modified after sensitive data have been provisioned in a previous execution, SANCTUARY's secure storage mechanism validates the integrity of the binary before enabling access to the stored data, as described below.

3.2.5.2 *Secure Storage*

SANCTUARY provides secure storage functionality to SAs, which is rooted in the secure world. The cryptographic keys used for secure storage are derived from the SA's binary integrity measurement, i. e., cryptographic hash of the binary, and a secret exclusively available to SANCTUARY's TCB, i. e., the platform key. Thus, only a SANCTUARY Instance started with an unmodified binary will allow the derivation of the correct key to decrypt previously stored data. Furthermore, by incorporating a unique platform key, stored data can be bound to a specific device, and by utilizing features of the secure world, e. g., monotonic counters, roll-back protection can be achieved.

3.2.5.3 *Code and Data Isolation*

SANCTUARY enforces code and data isolation through the same strong physical memory partitioning mechanism that is used to separate TrustZone's normal world and secure world. Only the CPU core that is assigned to the SANCTUARY Instance can access the reserved SANCTUARY memory, hence, memory isolation is ensured. SANCTUARY's memory isolation is configured by the TF and is enabled before the integrity of loaded code, i. e., SL and SA, is verified, and also before any confidential data is loaded, e. g., from secure storage. Hence, once the integrity of the loaded code is validated, its integrity

is preserved by the isolation. Similarly, data confidentiality is preserved as critical data is only made available to an SA when isolation is enforced. Before a SANCTUARY Instance is terminated all confidential data is erased before deactivating the memory isolation, hence, when the memory is made available to the LOS again, no confidential data leaks. To prevent the leakage of confidential data through CPU registers, e. g., during context switches, SANCTUARY ensures that the selected CPU core cannot change execution mode in an uncontrolled way. In particular, before confidential data is processed, SL configures the interrupt handling of the SANCTUARY CPU core such that it cannot be influenced from untrusted devices or other CPU cores. This configuration cannot be reverted by another CPU core executing potentially malicious code. This also prevents that the SANCTUARY CPU core can be forcefully shutdown.

SANCTUARY Instance can selectively share information with trusted parties, e. g., SANCTUARY services such as secure storage. When initiating a request to SANCTUARY's secure services a SANCTUARY Instance issues a Secure Monitor Call (SMC), which causes a switch to the TF before control is transferred to the security service executing in the secure world. The TF validates the origin of the call and passes this information to the security service; thus, the security service can authenticate the entity that initiated the request. This prevents that security services can be misused in confused deputy attacks.

3.2.5.4 *Cache Attack Protection*

SANCTUARY prevents cache-based attacks by ensuring that caches used by a SANCTUARY Instance are excursively used by this SANCTUARY Instance.

For CPU core exclusive caches, e. g., the L1 cache on typical ARMv8 platforms, the adversary is left with only two options. (1) Influence the cache state before the CPU core is assigned to a SANCTUARY Instance in order to observe changes in the SA's behavior, such as changed execution time. This is prevented by setting the cache into a defined state before executing a SANCTUARY Instance reverting all influences by the adversary. (2) After a SANCTUARY Instance finished execution the adversary can try to extract information remaining in the cache. This is prevented by flushing the cache before exiting a SANCTUARY Instance.

Caches shared between CPU cores (e. g., L2 cache on ARMv8) have the potential to leak sensitive information from a SANCTUARY Instance when used simultaneously by SAs and untrusted software. To prevent such leakage SANCTUARY ensures that shared caches are not used by the SA by configuring the caching policy for the isolated memory accordingly.

3.2.5.5 *Malicious Sanctuary App*

Sanctuary Apps (SAs) can be malicious and an adversary can try to misuse them to gain privileged control over the system. An adversary controlling only unprivileged normal-world code and an SA can aim to compromise the LOS or attack other LAs. SAs have, similar to LAs, unprivileged execution permissions (`EL0`). The privileged execution mode in a SANCTUARY Instance is controlled by the SL, which is provided by the device vendor and can therefore be trusted to not exploit its privileges to compromise the

normal-world software or disturb the system's operation. Furthermore, SANCTUARY's memory isolation prevents the SL from directly accessing the LOS's memory and memory of LAs that is not explicitly shared. Hence, an adversary does not gain an advantage from controlling a malicious SA compered to control over a malicious LA, i. e., in both cases the adversary has to find and exploit a vulnerability in the underlying privileged software (SL or LOS respectively) to escalate its privileges.

To attack the secure world control over a SANCTUARY Instances does not provide the adversary with additional capabilities. SANCTUARY Instance execute in normal-world mode and have no additional privileges compared to the legacy software, such as the LOS and LAs, executing in the normal world.

3.2.5.6 *Security Requirements*

Requirement 1: *Code and data integrity* is fulfilled by the combined properties described in Section 3.2.5.1 and Section 3.2.5.3. SANCTUARY fulfills requirement 2: *Data confidentiality* through the combination of the properties described in Section 3.2.5.2, Section 3.2.5.3 and Section 3.2.5.4. Requirement 3: *Secure channel to secure world* is fulfilled by the properties described in Section 3.2.5.3. Requirement 4: *Protection from malicious SAs* is fulfilled as described in Section 3.2.5.5. And requirement 5: *Hardware-enforced resource partitioning* is fulfilled by SANCTUARY's properties described in Section 3.2.5.3 and Section 3.2.5.4.

3.2.6 *Evaluation*

We evaluated different aspects of SANCTUARY using our prototype implementation (cf. Section 3.2.4) on the HiKey 960 development board. Firstly, our prototype implementation adds less than 1400 LoC to the system's TCB. The end-to-end performance of SANCTUARY is evaluated using a real-world use-case prototype. Our prototype implements a typical security critical functionality: a one-time password generator as used, for instance, in online banking. It shows that SANCTUARY does not disrupt the LOS while SAs execute. Furthermore, all components were evaluated individually – using micro-benchmarks – as well as the complete life cycle of an SA to demonstrate the exact run-time behavior of SANCTUARY. Our evaluation shows that a SANCTUARY Instance can be launched in less than 200 ms, where 7 ms are required to load the SA binary, 113 ms are consumed to shut down and remove one CPU core from the LOS's control, 13 ms are required by the secure world to verify the loaded code and apply the memory isolation policy, and 59 ms are required to start up the isolated SANCTUARY core with both, the SL and the SA. Further, typical operations, such as communications between an SA and a TA, are evaluated.

A detailed description of our evaluation setup, our evaluation methodology as well as our extensive evaluation results are presented in our conference publication [73].

3.2.7 *Conclusion*

SANCTUARY is novel security architecture for extending the TrustZone software ecosystem with user-space enclaves, called Sanctuary Apps (SAs). SANCTUARY provides hardware-enforced two-way isolation obviating the need to trust or vet the code of SAs, as malicious SAs have no more privileges or capabilities than normal user-space applications.

SANCTUARY is based on the bus master identity filtering feature, which was introduced with ARM's TZC-400 memory controller design and allows the parallel isolation of individual CPU cores for executing security-sensitive code. Furthermore, our proof-of-concept implementation and our evaluation results show SANCTUARY's low latency for typical use cases, all of which constitutes to the high practicality of our SANCTUARY security architecture.

3.3 Related Work

A wide variety of security architectures has been developed by industry as well as academia. These security architectures target different platforms, ranging from server and desktop computers to low-end embedded systems, pursue different design goals, use varying approaches and mechanisms, and operate under diverse assumptions.

TyTAN and Sanctuary, as most hardware security solutions, provide isolated execution environments that integrate with the System-on-Chip (SoC), which enables Trusted Execution Environments (TEEs) that can utilize the computing capabilities of the main processor while executing in isolation. In contrast, solutions such as the Trusted Platform Module (TPM) [383] provide a dedicated co-processor that usually has limited (computation) resources and limited functionality.

Integrated TEEs can be realized using various memory isolation mechanism, where the available mechanisms depend on the underlying platform and architecture. Systems can be divided into platforms that support virtual memory abstractions, as common in end-user and high-end devices such as smartphones, desktop and laptop computers or servers, and platform where software operates using physical memory addresses, commonly used in low-end devices such as micro-controllers (cf. Section 2.1.1).

Solutions for both types of systems have been proposed in the literature. We discuss them subsequently. Virtual-memory-based systems (Section 3.3.1) are most related to Sanctuary, as it targets ARM systems with virtual memory support, while TyTAN is closer related to security architectures for physical-memory-bases systems (Section 3.3.2).

Maene et al. [259] compare different *Trusted Computing Architectures*, including TyTAN, with respect to seven security properties (support for isolation, attestation, sealing, Dynamic Root of Trust (RoT), code confidentiality, side-channel resistance, and memory protection) as well as seven architectural properties (lightweight, co-processor-based, hardware-only Trusted Computing Base (TCB), preemption of enclaves, upgrade-able TCB, and backwards compatibility). With respect to architectural properties they find that TyTAN does not rely on a co-processor, and that TyTAN's TCB consists of hardware and software. Regarding security properties, the authors conclude that TyTAN is lacking support for code confidentially, side-channel resistance and memory protection, i. e., encryption of data stored to memory outside the SoC. While TyTAN does not provide code confidentially by default, it does provide a secure provisioning mechanism, thus loading of confidential code can trivially be implemented by a secure task if need. TyTAN does not provide explicit protection against side-channel attacks, as TyTAN's target platform has neither caches nor virtual memory management. It is thus inherently secure against the side-channels listed by Maene et al. [259]. Similarly, TyTAN's target platform uses on-chip Static Random Access Memory (SRAM), i. e., it does not use memory outside of the SoC containing sensitive data that would need to be protected. Sanctuary is not considered in the work by Maene et al. [259] as it was published later.

Strackx and Piessens [367] propose the Ariadne framework, addressing the state-continuity problem, which is relevant for many enclaved applications. Relying on secure non-volatile memory, Ariadne can provide state-continuity, including roll-back protection while also ensuring that an enclave interruption, e. g., due to a system crash, does

not render an enclave inoperable. When extending our security solution (TyTAN and Sanctuary) with secure non-volatile memory, the Ariadne framework can be used with our security solutions.

3.3.1 *Virtual-Memory-based Systems*

The most widely used processor architectures supporting virtual memory are Intel's x86 architecture [208] and ARM processors [29]. More recently, the RISC-V architecture has emerged as an open source Instruction Set Architecture (ISA) [403] with (optional) support for virtual memory. Security solutions for these systems are typically too complex and not applicable to low-end embedded systems, as targeted by TyTAN.

The security solutions for these systems either extend the platform with additional memory isolation mechanism in hardware (see Section 3.3.1.1) or utilize the memory isolation provided by the virtual memory mechanism (see Section 3.3.1.2).

3.3.1.1 *Hardware Security Architectures*

Intel Software Guard Extensions (SGX) [206, 267] and ARM TrustZone [24] are two widely deployed security solutions available in commercial off-the-shelf products.

Intel's SGX provides isolated execution environments, called *enclaves*, as described in more detail in Section 2.3.2. However, SGX is developed particularly for Intel's x86 architecture and therefore not available on other architectures that are common in embedded and mobile devices.

For ARM TrustZone [24], strict vendor policies regarding which Trusted Apps (TAs) get access to the secure world limit the availability of TrustZone's security functions to third party developers, as discussed in Section 2.3.1. Sanctuary overcomes this restriction by providing isolated enclaves in the normal world that can run arbitrary sensitive code while limiting the secure-world code to a minimal and fixed set of functionalities.

SecTEE proposes an enclave architecture for systems with ARM TrustZone, where enclaves are executed as trusted applications inside the secure world [422]. To protect enclaves from physical attacks on the Dynamic Random Access Memory (DRAM), SecTEE uses the On-Chip Memory (OCM) mechanism of OP-TEE [246] to store enclave's code and data during execution in on-chip memory only. Further, SecTEE extends the memory management of the secure world, to implement cache coloring, prevent cross-enclave side-channel attacks. To protect against side-channel attacks from the normal world, SecTEE uses cache pre-loading based on techniques from Zhang et al. [417]. SecTEE core mechanism, i. e., software-isolated enclaves in the secure world, is similar to open source systems such as OP-TEE [245] and commercial solutions such as Qualcomm's SEE that has been shown to be vulnerable to software attacks [49] (see Section 2.3.1 for a more detailed discussion on the limitations of TrustZone's software-only isolation approach). Sanctuary, in contrast, uses the same strong hardware-based isolation that separates the secure world from the normal world to isolate the system's TCB from potential malicious enclaves.

Further security architectures proposed by academic research [110, 369, 90] require hardware modification hindering their adaption in practice or have negative impact on the system's overall performance [264, 371].

Sanctum [110] proposes a security architecture for RISC-V platforms that provides enclave execution similar to Intel's SGX. In addition, Sanctum offers protection against side-channel attacks. It integrates cache partitioning for the Last Level Cache (LLC) and flushes the per-code L1 cache upon enclave exit. Furthermore, each enclave is responsible for its own virtual memory management (paging).

AEGIS [369] is a security architecture that can provide temper-evident execution of programs, guaranteeing the correct execution of a program, and private temper-resistant execution, which additionally ensures the confidentiality of a program's data. AEGIS' TCB can optionally be implemented completely in hardware or in a software-hardware co-design where parts of the TCB functionality is implemented as a secure kernel in software.

Flicker [264] and TrustICE [371] provide *temporal isolation* only, i. e., they cannot provide isolation for systems where TEEs execute in parallel with untrusted software. Hence, on multi-core systems – as commonly used today – the applicability of these approaches is very limited. With temporal isolation, the entire system has to be suspended, i. e., hibernation of the entire untrusted software stack including Operating System (OS) and all applications. Afterwards, the TEE can execute exclusively on the system; only after the TEE has terminated, the normal system can be restored and continue execution. Flicker uses Intel's Trusted Execution Technology (TXT) to reset the system at run time to a trusted execution state. TrustICE is conceptually similar to Flicker: it uses the TrustZone secure world, rather than TXT, to reset the normal world to a trusted state prior execution of sensitive code.

3.3.1.2 *Software Security Architectures*

Isolation of TEE instances can be implemented by trusting a privileged software entity, e. g., a hypervisor or micro-kernel, that configures and manages the (extended) virtual memory system of existing hardware platforms [90, 266, 197, 35, 140]. In these systems the hypervisor (or micro-kernel) is part of the TCB and its integrity is crucial for the system's security, hence, its integrity must be guaranteed by existing hardware mechanism [266, 197, 140] or by custom hardware extensions [90]. However, the usage of a hypervisor or micro-kernels results in a relatively large and complex TCB, which makes ensuring the TCB's correctness a non-trivial task [140, 141]. In addition, most solutions occupy the hardware-virtualization functionality of the platform prohibiting its use in scenarios where these features are required for non-security functions.

Basion [90] utilized hardware-supported virtualization controlled by a trusted hypervisor to create and manage isolated execution environments. The integrity of the hypervisor itself is ensured via a secure boot mechanism.

TrustVisor [266] uses a hypervisor to extend Flicker's concept [264] to provide isolated execution environments while improving flexibility and performance. TrustVisor uses the

Dynamic Root of Trust for Measurement (DRTM) [9] concept to ensure the hypervisor's integrity.

vTZ [197] provides virtualization-based TEEs in the normal world of ARM TrustZone enabled devices. The integrity of vTZ's hypervisor is ensured by extension of TrustZone's secure boot mechanism.

Secure Kernel-level Execution Environment (SKEE) [35] creates an isolated execution environment *within* the kernel in order to perform additional policy checks on kernel operations.

HYDRA strives to reduce the hardware features required for Remote Attestation (RA) architectures by utilizing formally verified software [140, 141]. In particular, HYDRA utilizes the formally verified seL4 [223] to realize memory access control and protection, which are required to protect the attestation routine and an authentication secret. Load-time integrity of the software TCB is ensured via secure boot and verified boot, respectively. During run time integrity is guaranteed by the memory access control based on a standard Memory Management Unit (MMU), as commonly available on more capable devices that are targeted by HYDRA. TyTAN targets smaller embedded systems, which do not provide memory virtualization, i. e., provide an MMU. However, the base concepts of TyTAN and HYDRA are similar, i. e., a trusted software component is responsible for (re-)configuring a memory access control hardware mechanism at run time. TyTAN aims to minimize the size and complexity of the trusted software while HYDRA relies on formal verification to ensure the correctness of the software TCB.

3.3.2 *Physical-Memory-based Systems*

Security solutions for small embedded systems, i. e., systems without virtual memory abstraction, are typically based on hardware-enforced isolation of security-critical code and data from other software on the same platform. Most solutions aim to provide TEEs for arbitrary programs [368, 286, 125, 287, 173, 226, 30, 406], some without assuming hardware-based isolation mechanism [120, 19], while others provide tailored solutions to provide protection of specific functions, such as RA [139, 147]. However, unlike TyTAN, none of these approaches can fulfill all requirements for a real-time system as identified by Stankovic and Rajkumar [362], e. g., bounded execution time for all primitives.

3.3.2.1 *Hardware Security Architectures*

Self-protecting modules (SPM) is a concept that extends an OpenMSP430 embedded processor with an Execution-Aware Memory Protection Unit (EA-MPU), which can be utilized by three new processor instructions to create isolated tasks [368]. However, these isolated tasks have a fixed memory layout and cannot be interrupted for preemptive scheduling as required for real-time systems [362]. SPM has been extended to support interrupts for isolated tasks [125] and extended with remote management capabilities (e. g., secure provisioning) [286, 287] as well as confidentiality for loaded code [173]. However, the execution time of the integrity measurement operation performed for newly created tasks remains unbounded, violating the requirements for real-time systems [362].

TrustLite generalizes the concept of code-dependent memory access control [226], as used by SPM [368] as well as SMART [139], by introducing a general purpose Execution-Aware Memory Protection Unit (EA-MPU). In addition, TrustLite supports interrupts for isolated tasks. However, TrustLite requires all software components to be loaded and their isolation to be configured at boot time. Further, the Inter-Process Communication (IPC) mechanism proposed by TrustLite is not applicable to dynamically loadable tasks.

TrustZone-M is a security architecture for low-end ARM processors [30]. Similar to the legacy TrustZone design, known from more complex processors, it partitions the system into two virtual worlds (known as *normal world* and *secure world*), where each world comprises a privileged and an unprivileged execution mode. In order to allow parallel execution of multiple sensitive tasks, the secure world needs to be partitioned by software, i. e., an OS, leading to a large and complex TCB.

Tan et al. [374] propose a systematic design approach for heterogeneous multiprocessor System-on-Chips (MPSoCs) implemented as Network-on-Chip (NoC). Their goal is to isolate tasks and restrict the accesses of tasks to resources, assuming that each task is directly mapped to a dedicated processing node, i. e., Tan et al. [374] are concerned with isolation at the NoC level rather than providing isolation of tasks sharing the hardware resources such as the Central Processing Unit (CPU) and memory.

SMART provides DRTM for embedded systems using a hardware/software co-design [139]. It protects the integrity of only one specific task, which is responsible for calculating RA reports, using Read-Only Memory (ROM). It ensures the confidentiality of the RA task's secrets through an extended memory access logic. SMART does not allow code changes after deployment and the integrity protected task must not be interrupted during attestation, rendering SMART incompatible for real-time systems.

TyTAN provides higher flexibility by providing dynamic loading and unloading of multiple tasks at run time, secure IPC with sender and receiver authentication, and real-time scheduling.

Malenko and Baunach [262] propose an approach for task isolation and exclusive peripheral usage for RISC-V micro-controllers without virtual memory abstraction. They sandbox tasks by utilizing the Memory Protection Unit (MPU) to ensure that tasks can only access their own memory and shared memory regions. To ensure that a task's memory is continuous in memory they modified the linker used in the task development tool chain. Furthermore, the MPU is extended to protect accesses to peripheral's memory mapped configuration registers on a per-task basis. This enables the OS to exclusively assign peripherals to tasks (at run time). Unlike TyTAN and Sanctuary, this approach does not aim to enable enclave-like execution, which requires, for instance, means to ensure the initial integrity of loaded software. Furthermore, the authors assume a trusted OS, which can access task memory (possibly by re-configuring the MPU).

SIA aims to provide a security architecture for intermittent computing device [132], i. e., devices that only operate if sufficient power is available using energy harvesting techniques. It utilizes Texas Instruments *Intellectual Property Encapsulation* extension available on some MSP430 systems [376], which allows to set code and data regions of the flash memory to be only accessible by the code in a specified code region. SIA utilizes this feature to isolate software modules on the platform, including security functionalities

such as RA. Unlike TyTAN and Sanctuary this approach provides only a single isolated entity on the platform, due to the fact that the Intellectual Property Encapsulation extension supports only a single isolated memory region. Furthermore, SIA assumes that the isolated code runs uninterrupted preventing its use in real-time systems.

3.3.2.2 *Software Security Architectures*

Daniels et al. [120] introduce SμV, a pure software-based isolation architecture for small embedded systems without hardware-based memory isolation capabilities, such as an MPU. A so-called microvisor, which was later formally verified [19], serves as the RoT, implementing memory isolation, and functionalities such as RA. Memory isolation is achieved by restricting assembly instruction available to untrusted applications and memory addresses accessed by untrusted application code. The microvisor enforces these restrictions by verifying all code that is loaded on the platform before execution. Additionally, SμV was extended to be compliant with the Global Platform [167] TEE standard specifications [212]. Unlike TyTAN, SμV is not concerned with real-time execution and does not support dynamic loading of sensitive applications.

4 TRUSTED EXECUTION ENVIRONMENT ATTACKS AND DEFENSES

Trusted Execution Environments (TEEs) are designed to protect the confidentiality and integrity of data processed and code executed inside of them. This is achieved by isolating the TEE from accesses by untrusted components and entities. Typically, this isolation is accomplished by employing additional access control rules on system resources, in particular memory. However, preventing *direct* access to memory is not necessarily sufficient to ensure isolation. In particular, side-channel attacks endanger the confidentiality of data processed in TEEs [412, 340, 277, 174] (cf. Section 2.4). In Section 4.1 we show that Intel Software Guard Extensions (SGX) is susceptible to cache side-channel attack. An adversary can exploit the fact that all system resources are under the adversary's control, aggravating the severity of side-channel attacks in the context of SGX.

Our attack as well as other published attacks [412, 389, 354, 183, 176, 174, 184, 405] against SGX demonstrate that enclaves need additional protection against side-channel attacks. In Section 4.2 we present HardIDX, a secure searchable encryption solution utilizing Intel SGX. It enables search over outsourced encrypted data, based on a database index with logarithmic size, outperforming all cryptographic searchable encryption solutions. HardIDX implements only the security critical core, i. e., the search engine, in an enclave while the data are stored outside of the enclave. While this approach benefits HardIDX's scalability, it induces extensive accesses to external resources, leading to potential information leakage. HardIDX protects against such leakage by randomizing external memory accesses and storage order. Furthermore, HardIDX provides protection against side-channel attacks leaking information at memory page granularity [412, 389, 354, 183, 176].

DR.SGX provides a software-only approach to protect SGX enclaves against side-channel leakage for *all* data access of an enclave to RAM (cf. Section 4.3). DR.SGX continuously re-randomizes the data memory layout of an SGX enclave to break the link between accessed memory locations – which are observable by an adversary – and the semantic of the data stored at that memory location. Without this semantic information the adversary cannot reconstruct which data has been accessed, and therefore, cannot deduce confidential information.

4.1 Software Grand Exposure: SGX Cache Attacks Are Practical

Intel Software Guard Extensions (SGX) promises integrity and confidentiality for data processed inside enclaves. However, shortly after the release of SGX attacks have shown that information about an enclave's memory access patterns can be leaked by exploiting the Operating System (OS)'s privilege to induce and observe page faults for an enclave [412].

In response to this class of attacks, different defenses have been developed that primarily aim at detecting the effects of such attacks [353, 96]. In particular, these defenses allow enclaves to detect whether they are interrupted frequently, e. g., due to page faults induced by an adversary. Thus, an enclave can avert the processing of sensitive information when (potentially) under attack.

However, it has also been speculated that SGX may be vulnerable to other side-channel attacks. In particular, cache-based side-channel attacks, which have been studied extensively independent of SGX [308, 294, 210, 252, 414, 180, 181], hold the potential to circumvent SGX's isolation guarantees.

Parallel with our work, multiple cache-based side-channel attacks targeting SGX enclaves have been developed [340, 277, 174]. We discuss the differences to our attack in Section 4.4.1.

Goals and Contributions. Our goal is a novel cache attack technique specifically targeted at SGX enclaves leveraging the full potential of SGX's adversary model, i. e., assuming the system's privileged software, such as the OS, being under the adversary's control. Our side-channel attack should be stealthy to prevent its detection by side-channel defense mechanism developed for SGX, in particular T-SGX [353] or Déjá Vu [96].

Furthermore, our goal is to show that not only cryptographic secrets and algorithms are susceptible to side-channel attack. Other sensitive data, such as genetic information, processed inside an enclave can be extracted via side-channel attacks, as well.

We make the following contributions:

- We introduce a novel cache side-channel attack technique against SGX enclaves. Our attack leverages the adversary's capabilities in the context of SGX to extract information from an enclave without interrupting the enclave, i. e., making the attack stealthy, and with low noise.
- We demonstrate that non-cryptographic applications leak highly sensitive information via side channels. In our case study on an enclave processing genome data we demonstrate that we can extract sufficient information to identify the person whose DNA is processed in the attacked enclave.

4.1.1 *Background*

This section provides background on Performance-Monitoring Counters (PMCs), which we use in our attack (Section 4.1.3). Background on Intel SGX as well as the cache architecture of Intel CPUs is provided in Section 2.3.2 and Section 2.1.2 respectively.

Performance Counter Monitor. The Performance Counter Monitor (PCM) is a feature of the Central Processing Unit (CPU) for recording hardware events. The PCM's primary goal is to give software developers insight into their program's effects on the hardware, allowing them to optimize their programs.

The CPU provides a set of PMCs, which can be configured to monitor different events, for instance, executed cycles, cache hits or cache misses for the CPU's different caches, mis-predicted branches. PMCs are configured by selecting the event to monitor as well as the mode of operation. This is done by writing to Model-Specific Registers (MSRs), which can only be done by privileged software. PMC are read via the `RDPMC` instruction (read Performance-Monitoring Counter), which can be configured to be available in unprivileged mode.

Hardware events recorded by PMCs could be misused as side channels, e. g., to monitor cache hits or misses of a victim process or enclave. Therefore, SGX enclaves can disable PMCs on enclave-entry by activating a feature called Anti Side-Channel Interference (ASCI) [208]. It suppresses all thread-specific performance monitoring, except for fixed cycle counters. Hence, hardware events triggered by an enclave cannot be monitored through the PMC feature. For instance, cache misses of memory loaded by an enclave will not be recorded in the PMCs.

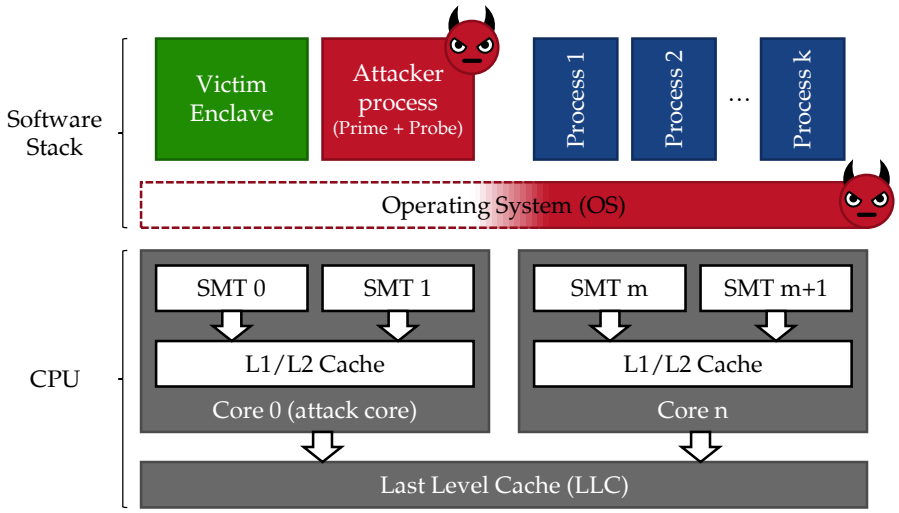
4.1.2 *System and Adversary Model*

We assume a system equipped with Intel SGX, i. e., a hardware mechanism to isolate data and execution of a software component from the rest of the system's software that is considered untrusted. The resources which are used to execute the isolated component (or enclave), however, are shared with the untrusted software on the system. The system's resources are managed by untrusted, privileged software, i. e., the Operating System (OS). Figure 22 shows an abstract view of the adversary model, an enclave executing on a system with an adversary-controlled OS, sharing a CPU core with an attacker process.

The adversary's objective is to learn secret information from the enclave, e. g., a secret key generated *inside* the enclave through a hardware Random Number Generator (RNG), or sensitive data supplied to the enclave *after* initialization through a secure channel.

Adversary capabilities. The adversary is in control of all system software, except for the software executed inside the enclave.[1] Although the adversary cannot control the program inside the enclave, the initial state of the enclave is known to the adversary, i. e.,

1 Due to integrity verification, the adversary cannot modify the software executed inside an enclave without being detected. Tempering by the adversary would be revealed by SGX's Remote Attestation (RA) feature, preventing that confidential data is provisioned to the enclave. SGX's secure storage feature prevents that a modified enclave can gain access to secrete data stored by a previous, benign instance of an enclave.



SMT: Simultaneous multithreading

Figure 22: High-level view of our attack; the victim enclave and the adversary's Prime+Probe code run in parallel on a dedicated CPU core. The malicious OS ensures that no other code shares that CPU core minimizing noise in the core-exclusive L1/L2 cache.

the program code of the enclave and its initial data. In particular, randomization through mechanisms such as Address Space Layout Randomization (ASLR) are visible to the adversary. The adversary knows the mapping of memory addresses to cache lines and can reinitialize the enclave and replay inputs, hence, the adversary can run the enclave arbitrarily often. Further, since the adversary has control over the OS and controls the allocation of resources to the enclave, including the time of execution, and the processing unit (CPU core) the enclave is running on. Similarly, the adversary can configure the system's hardware arbitrarily, e. g., define the system's behavior on interrupts, or set the frequency of timers. However, the adversary cannot directly access the memory of an enclave. Moreover, the adversary cannot retrieve the register state of an enclave, neither during the enclave's execution nor when an enclave is interrupted, and control is passed to the OS.

Adversary Goals. The adversary aims to learn about a victim enclave's cache usage by observing effects on the cache's availability to its own program. In particular, the adversary leverages the knowledge of the mapping of cache lines to memory locations in order to infer information about access patterns of an enclave to secret-dependent memory locations, which in turn allows the adversary to draw conclusions about sensitive data processed by a victim enclave. We show two concrete attacks for recovering a Rivest–Shamir–Adleman (RSA) key in Section 4.1.4.1 and identifying individuals in genome processing applications in Section 4.1.4.2.

4.1.3 *Our Attack Design*

Our attack technique is based on the Prime+Probe cache monitoring technique [294]. The "classical" variant of Prime+Probe is described in the Section 2.4. Subsequently we discuss our improvements of Prime+Probe beyond the basic approach.

4.1.3.1 *Prime+Probe for SGX*

Cache monitoring techniques, for instance Prime+Probe, experience significant noise. Therefore, most of the previously reported attacks, e. g., extracting a full cryptographic key, require thousands and even millions of repeated executions to average out the noise [415, 419]. Our goal is to build an efficient attack, i. e., an attack that works with a low number of repeated executions. The key to achieve this is to reduce noise (or pollution) in the cache monitoring channel.

There are two main aspects that guide our selection of possible noise reduction techniques – and also distinguish us from most of the previous attacks. (1) Our goal is to build an attack that cannot be easily detected using the early side-channel attack detection approaches for SGX [353, 96]; this requirement limits the possible noise reduction techniques we can use, e. g., we cannot interrupt the enclave's execution. (2) In contrast to "classical" side-channel attack the adversary is in control of the privileged OS; this condition enables us to leverage new methods that were previously inaccessible to the adversary (e. g., the use of PMCs).

Challenges. Given these conditions, we list the main challenges in our attack realization.

1. Minimizing cache pollution caused by other tasks.
2. Minimizing cache pollution by the victim itself.
3. Uninterrupted victim execution to counter side-channel protection techniques and to prevent cache pollution by the OS.
4. Reliably identifying cache evictions.
5. Performing cache monitoring at a high frequency.

Next, we describe a set of new attack techniques that we developed to address each of the challenges listed above.

4.1.3.2 *Noise Reduction Techniques*

(1.) Isolated Attack Core. We isolate the *attack core*, i. e., the CPU core on which the attack is executed, from all other processes executed on the system in order to minimize the noise in the side channel. Figure 22 shows our approach to isolate the victim enclave on a dedicated CPU core, which only executes the victim and the adversary's Prime+Probe code.

By default, Linux schedules all processes of a system to run on any available CPU core, hence, the caches of all CPU cores are affected all executed processes. The adversary cannot distinguish between cache evictions caused by the victim and those caused by any other process. By modifying the Linux scheduler, the adversary can make sure that attack core is exclusively used by the victim and the adversary ("Core 0" in Figure 22). This way no other process can pollute the attack core's exclusive caches, i. e., L1/L2 cache.

(2.) Self-pollution. The adversary needs to observe specific cache lines that correspond to memory locations relevant for the attack. From the adversary's point of view it is undesirable if those cache lines are used by the victim for any other reason than accessing these specific memory locations, e. g., by accessing unrelated data or code that map to the same cache line.

In our attack, we use the L1 cache, which has the advantage of being divided into a data cache (L1D) and an instruction cache (L1I). Therefore, code accesses, regardless of the code's memory location, never influence the cache lines of the L1 data cache, which is of interest to the adversary. However, victim accesses to unrelated data stored in memory locations that map to relevant cache lines lead to noise in the side channel. This noise source cannot be influenced by the adversary given that the memory layout of the victim is fixed. This challenge can be tackled partially by a high monitoring frequency (see below) in order to extract information before it is expunged by unrelated memory accesses.

(3.) Uninterrupted Execution. Interrupting the victim enclave yields two relevant problems. (a) When an enclave is interrupted, an Asynchronous Enclave Exit (AEX) is performed and the OS's Interrupt Service Routine (ISR) in invoked (see Section 2.3.2). Both, the AEX and the ISR use the cache, and hence, such an event induces significant noise in the side channel. (b) By means of transactional memory operations an enclave can detect that it has been interrupted. This feature has been used for a side-channel defense mechanism [353, 96]. We discuss the details of these defenses in Section 4.4.2. Hence, making the enclave execute uninterrupted is important for the victim enclave to remain unaware of our side-channel attack.

In order to observe the changes in the victim's cache throughout its execution, we need to monitor the cache of the attack core in parallel. To achieve this, we execute the attacker code on the same CPU core. The victim is running on the first Simultaneous Multithreading (SMT) execution unit while the attacker code is running on the second SMT execution unit (see Figure 22). As the victim and attacker code compete for the L1 cache, the adversary can observe the effects of the victim's execution on the cache.

The attacker code is, similar to the victim code, executed uninterrupted by the OS. Interrupts usually occur at a high frequency, e. g., due to arriving network packets, user input, etc. By default, interrupts are handled by all available CPU cores, including the attack core, and thus the victim and attacker code are likely to be interrupted. The OS code executed on arrival of an interrupt will pollute the cache, or the victim enclave could detect its interruption, assume an attack, and stop itself.

To overcome this problem, we leverage the adversary's control over the system's privileged software and configured the interrupt controller such that interrupts are not

delivered to the attack core, i. e., it can run uninterrupted. The timer interrupt is an exception and cannot be reconfigured the same way. Each CPU core has a dedicated timer and the interrupt generated by the timer can only be handled by the associated CPU core. However, we reduced the interrupt frequency of the timer to 100 Hz, which allows victim and attacker code to run for 10 ms uninterrupted. This time frame is sufficiently large to run a complete attack cycle undisturbed (with high probability).[2] As a result, the OS is not executed on the attack core while the attack is in progress (depicted by the dashed-line OS-box in Figure 22). Also, because the victim is not interrupted it cannot detect the attack with mechanism proposed in literature, such as T-SGX [353] or Déjá Vu [96].

(4.) Monitoring Cache Evictions. In the previous Prime+Probe attacks, the adversary determines the eviction of a cache line by measuring the time required for accessing memory locations that maps to the cache line of interest to the adversary. These timing-based measurements represent an additional source of noise to the side channel. Distinguishing between cache hit and miss requires precise time measurements. For instance, for the Skylake architecture Intel reports that a L1 cache a cache hit takes at least 4 cycles [205]. If the data got evicted from the L1 cache, they can still be present in the L2 cache. Reading data from L2 cache takes 12 cycles in the best case.This small difference in access times makes it challenging to distinguish a cache hit in L1 cache and a cache miss in L1 that is served from L2 cache. Furthermore, Intel states that "software-visible latency will vary depending on access patterns and other factors" [205]. Reading the time stamp counter to calculate the time a read operation takes, itself suffers from noise, which is in the order of the difference between L1 and L2 cache accesses. Thus, when the timing measurement does not allow for a definitive distinction between a cache hit and a cache miss, the observation has to be discarded. To eliminate this noise, we use PMCs to determine if a cache line got evicted by the victim. This is possible in the SGX adversary model because the adversary controls the OS and can freely configure and use the PMCs.

Usage of PMCs for cache attacks was previously explored by Uhsadel et al. [387], Bhattacharya and Mukhopadhyay [51]. For instance, Uhsadel et al. [387] demonstrated that measurements collected using PMC for L1 cache misses require the least amount of traces compare to other measurement methods, such as time stamp counters. One should, however, note that PMCs are only beneficial if adversary is privileged, but cannot directly read the victim memory. To the best of our knowledge, we are the first to use PMC in such a setting.

We recall that Intel processors provide the ASCI feature (cf. Section 4.1.1) that prevents monitoring of cache related events caused by enclaves' executions, i. e., the straightforward approach to monitor cache related events of the victim is obstructed by the fact that PMCs are disabled for enclave code (cf. Section 4.1.1). This, however, does not prevent our attack since our attack does not monitor cache activity of the victim. Instead, the adversary observes cache events of the attacker process, which shares the cache with the victim. In particular, the entire cache is primed by the adversary before the victim is executed. When the victim executes it evicts a subset of cache lines. In the probe phase

2 When an interrupt occurs by chance the attack can be repeated. If the time frame is too short the timer frequency can be reduced further.

the adversary detects which cache lines were evicted by monitoring cache miss events for the attacker process.

(5.) Monitoring Frequency. As discussed before, the victim should run uninterrupted while its cache accesses are monitored in parallel. Hence, our attack needs to execute priming and probing of the cache, i. e., sample the cache usage of the victim, at a high frequency to not miss relevant cache events. In particular, probing each cache line to decide whether it has been evicted by the victim is time consuming and leads to a reduced sampling rate. The required monitoring frequency depends on the frequency at which the victim is accessing the secret-dependent memory locations. To not miss any access, the adversary has to complete one prime and probe cycle before the next relevant access by the victim occurs. In our implementation the access to PMCs is the most expensive operation in the Prime+Probe cycle.

To tackle this challenge, we monitor individual (or a small subset of) cache lines over the course of multiple executions of the victim. In the first run, we learn the victim's accesses to the first cache line, in the second run accesses to the second cache line, and so on. By aligning the results of all runs we learn the complete cache access pattern of the victim.

4.1.4 *Attack Instantiations*

We implemented and evaluated our attack for two concrete targets. We performed and evaluated our attack on a Dell Latitude E5470 with an Intel Core i7-6600U CPU (at 2.60 GHz) running Ubuntu Linux 14.04 with a customized kernel in version 4.4.0-57 and Intel SGX Software Developer Kit (SDK) version 1.6.

RSA Attack. The first target for our attack was the RSA algorithm. Attacks on cryptographic algorithms are the prime example for many cache side-channel attacks, as these algorithms process highly sensitive information (i. e., the secret key). Therefore, attacks on cryptographic primitives are well studied in previous works [308, 294, 210, 252, 414, 180, 181]. This makes a cryptographic algorithm an obvious first attack target. We describe our attack subsequently in Section 4.1.4.1. Parallel works attacking cryptographic implementations in SGX [340, 277, 174] and their differences to our approach are discussed in Section 4.4.1.

Genome Sequence Attack. In our second attack instance, we target an enclave processing genome sequences, i. e., a non-cryptographic algorithm that nevertheless processes highly sensitive data.

Due to the fact that cryptographic algorithms are popular targets for side-channel attacks, many cryptographic libraries are hardened against cache monitoring using techniques such as scatter-gather [75]. Hence, side-channel attacks against cryptographic libraries can be thwarted by using appropriate implementations [50].

Unfortunately, the implementations of most other, non-cryptographic algorithms processing sensitive data are not hardened in a similar way, making them vulnerable to cache monitoring attacks. While these algorithms can be manually hardened as well, it requires both developer expertise and effort to make an algorithm side-channel resilient.

As a consequence, many non-cryptographic algorithms, used inside SGX enclaves to process sensitive data, remain vulnerable to cache monitoring attacks.

In Section 4.1.4.2 we demonstrate that critical information can be extracted from an SGX enclave processing genome data.

4.1.4.1 *RSA Attack*

In our first attack instance, we targeted an RSA implementation from the Intel Integrated Performance Primitives (IPP) crypto library, which is part of Intel's SGX SDK. The private key was chosen randomly. In particular, we extracted memory access patterns of the decryption algorithm from the fixed-size sliding window exponentiation implementation [207]. We implemented a victim enclave that uses this library to decrypt a single message when started. The enclave was compiled with default optimization flags and compiler settings. Intel's IPP crypto library also includes an RSA implementation variant that is hardened against cache attacks. However, the goal of this work is to show the effectiveness of our attack techniques against cryptographic algorithms, which are implemented without the use of explicit protection mechanism, to make our results comparable to previous and concurrent works on side-channel attacks. We discuss defenses in Section 4.4.2 and present our own side-channel defense DR.SGX in Section 4.3.

Our target implementation of RSA uses the Chinese Remainder Theorem (CRT) to optimize performance. A message is decrypted by separate exponentiation operations using two pre-computed values d_p and d_q. The values d_p and d_q are derived in advance from the cryptographic key's prime factors p and q, i. e., the private key. In our experiment we used RSA keys of length 2048 bit, hence, during the decryption exponentiation is performed two times, each being a 1024 bit exponentiation.

The implementation of the fixed window exponentiation of our target is shown in Figure 23. The algorithm takes the base a, the exponent e (i. e., d_p and d_q respectively), and the parameter N (which is public). First, the multiplier table g is pre-computed for the given a. Next, the exponent e is divided into a 2^k representation, i. e., the exponent is divided into windows of fixed size (k bit), resulting in $\lceil n/k \rceil$ windows $(e_j, e_{j-1}, \ldots, e_1, e_0)$. Calculating the exponentiation result involves a lookup in the table g for each window $(e_j, e_{j-1}, \ldots, e_1, e_0)$ determining the pre-calculated multiplier that has to be multiplied with the intermediate result. Hence, for each window one entire from the table g is accessed.

Figure 23 shows the lookup in the table g for the windows of e. Due to the size of the table entries, each entire spans two cache lines, i. e., the table access for the first window (`1010`) will affect cache lines 19 and 20.

Attack Details. Our attack employs all noise reduction mechanism described in Section 4.1.3. In particular, to increase the monitoring frequency we monitor accesses to each multiplier, i. e., entries in table g, in individual executions. Each multiplier is 1024 bit and occupies two, consecutive cache lines when accessed. Thus, to detect the use of a particular multiplier, we probe the two corresponding cache lines. Our attack allows us to make one observation every c cycles. Next, we divide all observed cache accesses into epochs of p probes each. The size of the pre-computed multiplier in table g means that 16

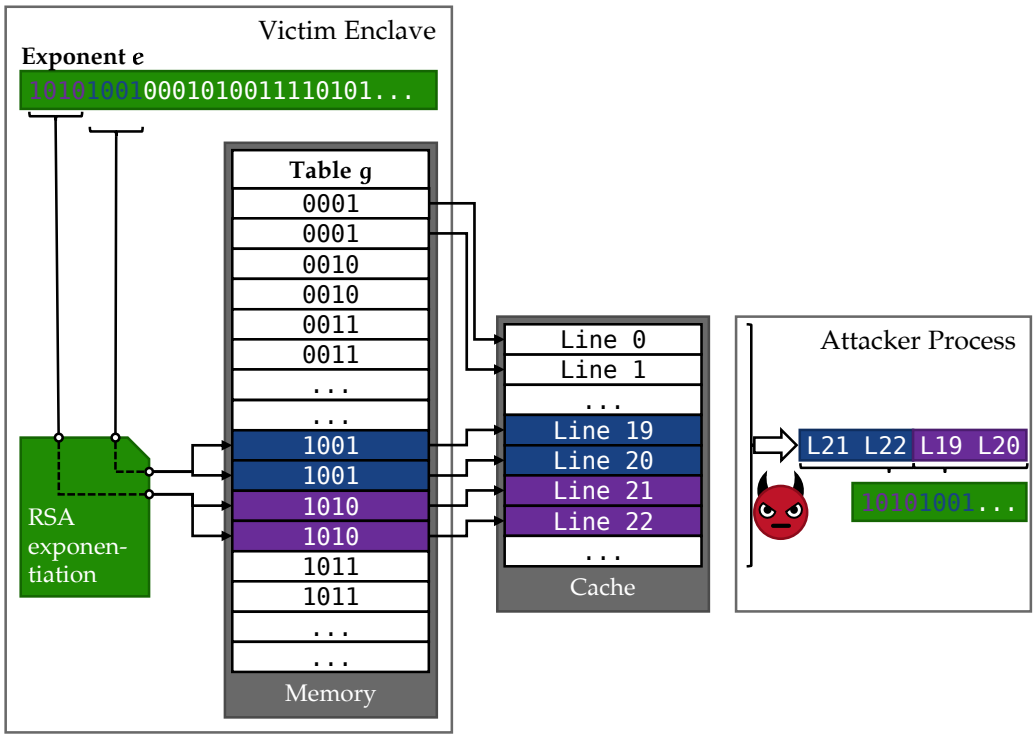


Figure 23: Memory accesses and cache updates in RSA exponentiation. The processed window value from exponent e determines the accessed entry from table g in memory which defines the impacted cache line.

repeated memory accesses are needed to read it from memory, i. e., we expect 16 memory access to the table entire of a multiplier within one epoch. When our observations meet this condition, we considered the corresponding multiplier a candidate.

The accesses to each multiplier in table g is monitored consecutively. In our attack we, monitored 10 out of 16 multipliers in the table g. On one hand, monitoring a subset of multipliers is sufficient to extract enough information about the key to recover the entire key. On the other hand, we encountered significant noise in our observations of the cache lines corresponding to the remaining multipliers. This noise rendered these observation worthless for our goal of recovering the private key.

We repeated the observation of each multiplier $t = 15$ times to collect a sufficiently large fraction of the key. Our monitoring frequency was $\frac{1}{c} = \frac{1}{500}$ cycles; we divided our monitoring trace into epochs of $p = 33$ probes each, which gave accurate results. The epoch length is based on the average execution time required for a single iteration of the exponentiation algorithm. We varied the number of multiplies being observed during a single execution of the victim enclave as well as the monitoring frequency. However, monitoring more than one multiplier as well as increasing the monitoring frequency ($c < 500$) causes significant noise in the measurements, i. e., it does not improve our attack.

Attack Results. Our measurement results are illustrated in Figure 24. Each row of the graph corresponds to one measurement trace for each of the ten multipliers we monitored,

i. e., our $t = 15$ repetitions resulted in 15 rows in the graph. Whenever two cache lines that correspond to a particular multiplier is access 16 times within one epoch a dot is plotted; every multiplier is represented by a different color. Thus, a vertical line of same-colored dots is a clear indication that the cache lines associated with one particular multiplier was accessed at the same time during all repetitions of the algorithm. This allows us to conclude which multiplier was used at this point in the algorithm.

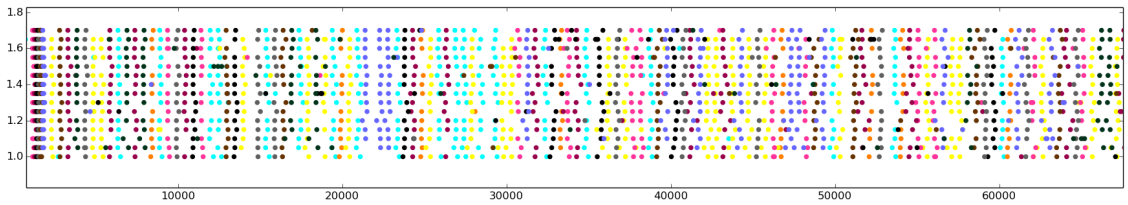


Figure 24: Access patterns to the table of pre-computed multipliers in the exponentiation of an RSA decryption. Each dot represents the access to the cache lines corresponding to one multiplier, i. e., 16 repeated memory accesses to particular cache lines. Each multiplier is plotted in a different color. All cache lines were monitored 15 times, each repetition is shown in a separate row. Vertical lines of same-colored dots clearly indicate the use of a particular multiplier.

In each iteration of the algorithm one multiplier is accessed, which produces one access observation in each repeated execution of the victim enclave, resulting in a vertical line in the graph (Figure 24). Since the execution time for the iterations of the algorithm is sufficiently constant, we can correlate each line in the measurement plot to its window of the exponent e.

In our experiment, we used a simple heuristic to decide whether a multiplier was accessed: if more than half of the monitoring rounds report the same value for a single epoch, we consider this value confirmed and assume that the respective multiplier was used by the algorithm. If no multiplier accesses are observed in an epoch, we conclude that the exponent window for this iteration was zero and no table entry was accessed.

Using this approach, we could extract approximately 70 % of all bit of our randomly chosen key correctly, which is sufficient information to reconstruct the entire key, and thus, rendering our attack successful.

4.1.4.2 *Genome Sequence Attack*

Genome data analysis is an emerging field that highly benefits from cloud computing due to the large amounts of data being processed. At the same time, genome data is highly sensitive, as they allow the identification of persons and carry information about a person's predisposition to specific diseases. Thus, maintaining the confidentiality of genomic data is paramount, in particular when processed in untrusted cloud environments.

Genome sequences are represented by the order of the four nucleotides: adenine, cytosine, guanine and thymine, usually abbreviated by their first letter (`A`, `C`, `G`, `T`). Microsatellites or Short Tandem Repeats (STR) are repetitive nucleotides base sequences. STR analysis is a common genomic forensics technique, where the lengths of the microsatellite (STR) at specific locations in the genome are used to identify individuals [78].

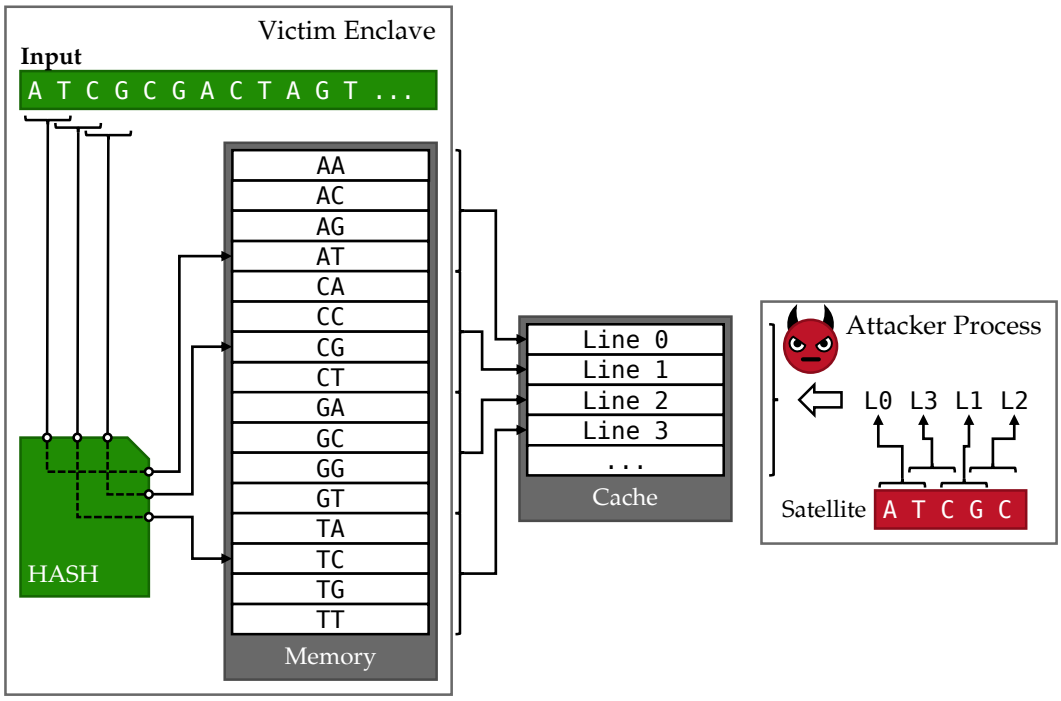


Figure 25: Genome sequence analysis based on hash tables; the positions of the genome's subsequences (called k-mers) are inserted into a hash table for statistical analysis and fast search for k-mers.

For example, many US forensics labs use STR lengths in 13 standardized locations to define a genotype for an individual.

Efficient search of genome sequences is vital for many analysis methods. Therefore, the genomic data is usually preprocessed before the actual analysis is performed. One common way of preprocessing is to divide the genome sequence into substrings of a fixed length k, called k-*mer*. The k-mers represent a sliding window over the input string of genome bases.

In Figure 25 the input sequence "`AGCGC...`" is split into 2-mers. Starting from the left the first 2-mer is `AG`, next the sliding window is moved by one character resulting in the second 2-mer `GC`, and so on. The k-mers are inserted into a hash table, usually, for each k-mer its position in the genome sequence is stored in the hash table. Thus, given a k-mer that is part of a microsatellite one can quickly lookup at which positions it appears in the input genome sequence.

Another use case is statistical analysis of the input genome sequence, for instance, the distribution of k-mers in the sequence can easily be extracted from the hash table.

Primex. Our victim enclave implements the preprocessing step for a genome sequence analysis, as described above. Our victim enclave uses an open source k-mer analysis tool called PRIMEX [241], which inserts each k-mer position into the hash table. Each hash table entry holds a pointer to an array that stores the positions of each k-mer.

Attack Details. Our attack aims at leaking the length of microsatellites (STR) at the standardized locations used for STR analysis when an input genome sequence is preprocessed (indexed) by the victim enclave. According to our attack assumptions (cf. Section 4.1.2) the adversary is in control of all system resources. This enables the adversary to prevent interference with the victim enclave's execution leading to a deterministic execution time for our victim enclave, allowing precisely correlating of cache monitoring observations with position in the input sequence.

Through our cache side channel, we can observe cache activities, which can be linked to the victim's insertion operation into the hash table. Figure 25 shows how insertions into the hash table affect different cache lines. For each k-mer the victim looks up a pointer to the respective array from the hash table. From the source code we learn the hash function used to determine the table index for each k-mer. By reversing the mapping function, we are able to infer the input based on the accessed table index.

Unfortunately, individual table entries do not map to unique cache lines. Multiple table entries fit within one cache line. Therefore, we cannot directly conclude which index was accessed from observing cache line accesses. This problem is illustrated in Figure 25. Here four table indexes map to a single cache line. When the adversary observes the eviction of cache line 0, the adversary does not learn the exact table index of the inserted k-mer. The adversary only learns a set of candidate k-mers that could have been inserted: `{AA,AC,AG,AT}`.

However, the adversary can divide a microsatellite of interest into k-mers and determine which cache lines will be used when it appears in the input sequence. In Figure 25 the microsatellite is split into four 2-mers, where the position of the first 2-mer (`AT`) will be inserted in the first quarter of the table, hence, cache line 0 will be used by the victim enclave. The position of the second 2-mer (`TC`) will be inserted into the last quarter of the hash table, thus activating cache line 3. Following this scheme, the adversary determines a sequence of cache lines that will be used by the enclave when processing the target k-mer. Monitoring for this sequence will reveal to the adversary at which position of the input genome the target k-mer is processed.

Attack Results. We provided a genome sequence string to the victim enclave and executed it in parallel to our attack code. We chose $k = 4$ for the k-mers leading to $4^4 = 256$ 4-mers (four nucleotides possible for each of the four position). Each 4-mer is represented by a unique table entry, each table entry is a pointer (8 Byte), and thus each cache line contains 64 Byte/8 Byte = 8 table entries.

The adversary's goal is to determine the microsatellite length at each of the 13 standardized locations. For example, in location CSF1PO the aim is to extract the length of the repeating sequence `TAGA`, which is expected at this position. First, the four 4-mers occurring repeatedly in the microsatellite are determined, as well as the corresponding cache lines for each 4-mer: `TAGA` ⇒ cache line 7; `AGAT` ⇒ cache line 28; `GATA` ⇒ cache line 9; `ATAG` ⇒ cache line 20.

In our attack, we monitor these four cache lines individually and align them, as shown in Figure 26. When the microsatellite appears in the input string, the cache lines 7, 28, 9 and 20 will be used repeatedly by the victim enclave. This increase in utilization of these cache sets can be observed in the measurements. In Figure 26, the increased density of

observed cache events is visible, marked by the solid line rectangle. Since all four cache lines are active at the same time, one can conclude that the microsatellite occurred in the input sequence.

False Positives. False positives due to monitoring noise are highly unlikely due to the fact that we observe four cache lines. Figure 26 shows extensive activation in the top cache line (pink) marked by the dashed line rectangle. However, in the three other cache lines there is low activity making this event clearly distinguishable from a true positive event, which is marked by the solid line rectangle as described before.

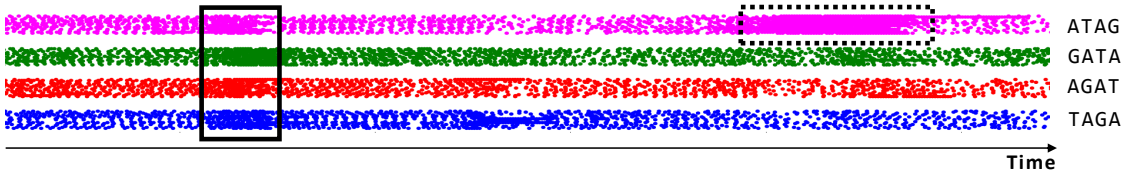


Figure 26: Access pattern of hash table accesses by PRIMEX processing a genome sequence [241]. Four cache sets are shown in different colors with 20 repeated measured for each cache set. The cache sets correspond to the 4-mers of the microsatellite `TAGA`. Increased activity in all four cache sets (marked by the solid line rectangle) indicates the occurrence of the microsatellite in the processed genome sequence.

Length Accuracy. For STR analysis one should, ideally, know the exact microsatellite length at sufficiently many standard locations. Due to cache monitoring noise, as seen in Figure 26, our attack cannot extract the precise length of microsatellites.

We determine possible microsatellite lengths by manually comparing the attack traces (see Figure 26) to pre-computed reference traces with known lengths. Through a manual verification we confirm that our attack is able to extract the length with an accuracy of ± 1 in the vast majority of test samples, i. e., if the true length is eleven in a location, e. g., CSF1PO, we learn from our measurements that microsatellite's length is between ten and twelve.

Target Identification. Given the three possible lengths for each standard location, one can compute the probability that a random person from a given population would match the leaked genomic information as the Combined Probability of Inclusion (CPI) [78]. The probability of identifying an individual depends on the genotype of the attack target. The worst case, and therefore the lower bound for our attack's accuracy, is when the target individual has the most common genotype, i. e., the most frequently seen microsatellite length in each of the 13 locations. Assuming that the attack target is a Caucasian person with the most common genotype. From the known frequencies [78, Chapter 11] we compute a CPI value of 7.027×10^{-5}. This means that from a population of 10 million people (e. g., a small country), statistically 703 people would have a genotype that matches the information leaked through our attack.

The average case is one where the attack target has more variation in his or her genotype. In the majority of such cases, statistically less than one person from a population of 10 million match the leaked information. We conclude that our attack is, therefore, able to identify the person whose genome is processed with high probability.

Other Applications. Similar information leakage is likely to apply to many other applications as well. Especially, programs that construct or access lookup tables, or similar data structures, are ideal targets for our attack. Such patterns are often seen in database systems, medical and scientific data processing applications, and machine learning models.

4.1.5 *Conclusion*

In this work we demonstrate that cache attacks on SGX are indeed practical and pose a serious threat on the core security benefit of SGX. Our goal was to develop an attack that cannot be mitigated by countermeasures aiming at detecting side-channel attacks [353, 96], therefore, we mount the attack on uninterrupted enclave execution. We developed a set of novel attack techniques and demonstrated our attack on RSA decryption and genome indexing.

4.2 HardIDX: Practical and Secure Index with SGX

Outsourcing encrypted data to untrusted cloud environments while still being able to search these data has been pursued for long time using cryptographic techniques. Encrypted databases have been approached using cryptographic schemes such as property-preserving encryption [44, 56], functional encryption [59] and searchable encryption [358, 117, 257]. However, for all these schemes performing efficient and secure *range queries* is challenging. Efficient methods, such as order-preserving encryption, e. g., used by CryptDB [314], are susceptible to simple attacks [282].

Most schemes for encrypted searches that support range queries have search time linear in the number of database entries, while schemes achieving polylogarithmic search time have query size, storage size or leakage problems [128, 144, 257]. Thus, designing an *efficient* searchable encryption scheme with *minimal leakage on the queried ranges* remains an open challenge.

General secure computation techniques, for example Secure Multiparty Computation (SMC) and Fully Homomorphic Encryption (FHE) [157], typically provide strong security guarantees, however, they are impractical for large-scale systems [158].

Another line of research leverages Trusted Execution Environment (TEE) technologies to enable secure databases [37, 41]. However, these solutions execute an entire unmodified Database Management System (DBMS) inside a TEE. And they do not formally consider the information leakage due to side channels or access patterns to resources outside of the TEE. Furthermore, they do not scale well as they do not account for the memory limitations TEEs typically face.

Goals and Contributions. Our goal is the design and implementation of an efficient and secure scheme for search over encrypted data that can be used as a database index. Our solution should provide high performance, significantly improving over existing software-based schemes [128, 144], by utilizing the isolation guarantees of Intel Software Guard Extensions (SGX), and provide better security and scalability, compared to previously proposed hardware-based solutions [37, 41]. Further, our scheme should organize data using B^+-tree structures, which are used as indexes in many DBMS [322].

The main contributions are as follows:

- HardIDX, our secure and efficient database index scheme. It has logarithmic complexity in the size of the index, allowing searches within a few milliseconds, even in large databases.
- We analyze HardIDX's security, in particular its information leakage, and show that it is comparable to the best known searchable encryption schemes.
- We provide an implementation and evaluate of HardIDX. We evaluate HardIDX for large databases with up to 50 000 000 key-value pairs and identify the performance and functional constrains of SGX.

4.2.1 *Background*

We provide background on SGX and side-channel attacks in Section 2.3.2 and Section 2.4, respectively. Subsequently we provide a brief introduction of B^+-trees.

B^+-tree. A B^+-tree is an n-ary tree, often used for efficient searches in databases. It has three types of nodes: a root node, internal nodes, and leaf nodes. It can store a large number of key-value pairs, where the keys are used to index the values. Values are only stored in leaf nodes.

B^+-trees used in storage systems often have a high fanout. Fanout denotes the number of child nodes each node can have, also referred to as *branching factor*.

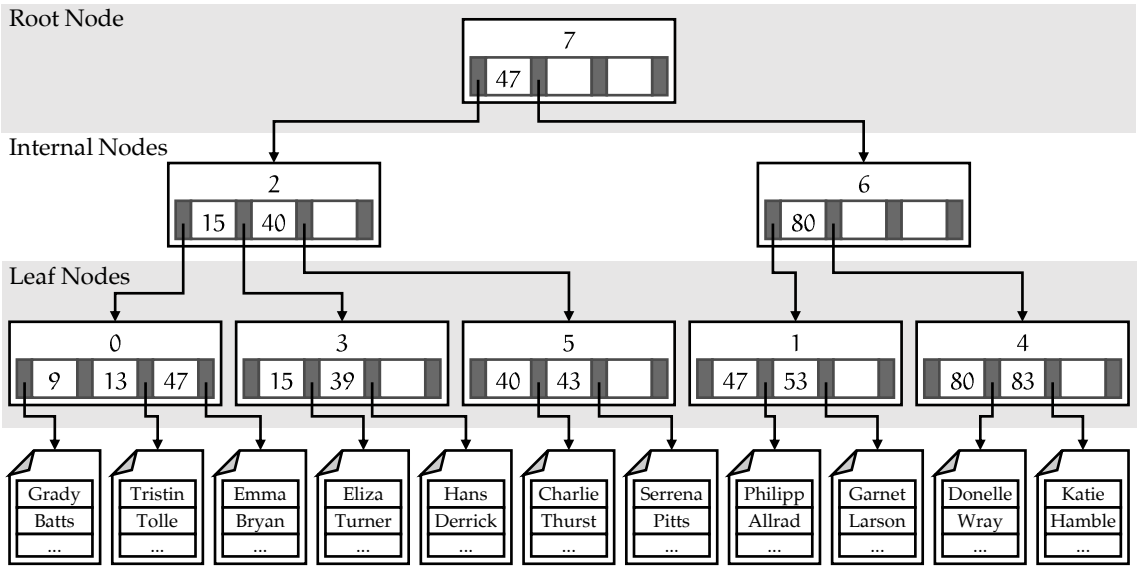


Figure 27: B^+-tree example: unique IDs for each employee are used as keys for the employee's record.

Figure 27 shows a B^+-tree for an employee database, where each value (record of employee data such as the name) is indexed by a unique search key (or ID). Every node x contains $|k|$ keys k, which are stored in a non-decreasing order: $k_1 \leqslant ... \leqslant k_{|k|}$ For every key k_i an internal node x_j holds a pointer p_i to the child node containing elements that are greater than or equal to k_i and smaller than all other keys $k_i + 1 \ldots k_{|k|}$ stored in x_j. p_0 points to a node which contains only elements smaller than k_1. In leaf nodes each pointer p_i points to the value corresponding to the pointer's key k_i.

4.2.2 *Model and Assumptions*

4.2.2.1 *System Model*

In this section we consider a setting in which a database is hosted on a server D[3] under the control of an untrusted service provider $\mathcal{SP}$ and serves queries (we focus on range queries) from a trusted client $\mathcal{C}$. Without loss of generality, we assume a single client $\mathcal{C}$ that is securely authenticated to the server D.[4]

In general, the server D, e. g., a cloud server, is untrusted. However, D has TEE capabilities, in particular, we assume a server equipped with Intel SGX. This enables D to provide an isolated execution environment, which we call HardIDX enclave. The HardIDX enclave is trusted by $\mathcal{C}$ after explicitly validating its integrity through Remote Attestation (RA). Furthermore, $\mathcal{C}$ validates, also through RA, that the latest microcode update for SGX is installed and Simultaneous Multithreading (SMT) is deactivated, making sure that the server is not vulnerable to a number of known side-channel attacks [390, 68, 174].

The initialization of the database is performed in a trusted environment. HardIDX focuses on the secure operation of the database, i. e., searches in the database.

4.2.2.2 *Adversary Model*

The adversary $\mathcal{ADV}$'s goal is to learn the content of the database stored on the server D or the content of the queries sent by the client $\mathcal{C}$. SGX's isolation prevents $\mathcal{ADV}$ from directly accessing the memory of an enclave. However, $\mathcal{ADV}$ aims to exploit information leaked through side channels to learn sensitive information.

$\mathcal{ADV}$ has full control over the server D's software, except for the HardIDX enclave. In particular, $\mathcal{ADV}$ controls D's privileged software, and thus, all resources of D. This enables $\mathcal{ADV}$ to (1) observe all interaction of the HardIDX enclave with system resources outside of the enclave, including the access pattern to (encrypted) data stored outside of the enclave. And (2) $\mathcal{ADV}$ can observe the memory usage of the enclave at the granularity of memory pages (4 kB), e. g., by utilizing a page-fault-based side-channel attack [412, 389, 354, 183, 176].

Cache side-channel attacks are considered out of scope, they are addressed in our side-channel defense DR.SGX (Section 4.3) as well as related works such as Cloak [182].

Hardware attacks are out of scope in HardIDX. Further, Denial-of-Service (DoS) attacks, e. g., on the cloud infrastructure or the network, are considered out of scope. In this section, we focus on passive attacks, strategies to prevent active attacks are presented in our journal article [151] and technical report [150].

3 For sake of consistency throughout this work all computing devices, whether they are servers or embedded devices, are named D.

4 We provide a discussion on multi-user setups in our conference publication [149].

4.2.3 *HardIDX Design*

The high-level design of HardIDX is shown in Figure 28. The design involves three entities: (1) the client $\mathcal{C}$ that is under the data owner's control and is therefore trusted. (2) The untrusted SGX enabled server D is under control of the untrusted service provider $\mathcal{SP}$ and (3) the trusted SGX enclave *within* the server D.

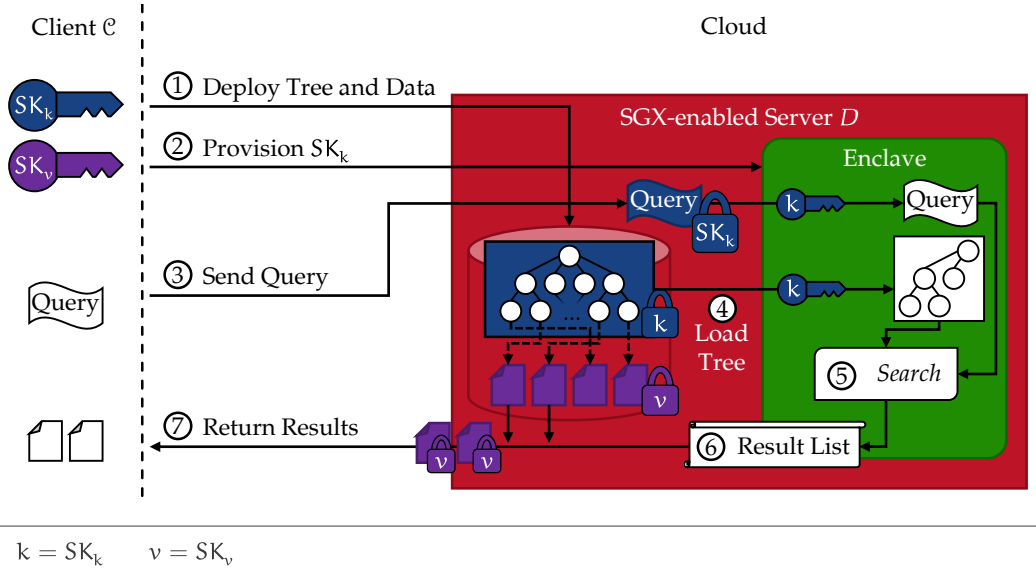


Figure 28: High-level design and life cycle of HardIDX. The search tree and data are stored encrypted in the cloud. Search queries are processed inside an enclave, which loads the search tree (partially) to perform the search. The encrypted results are sent back to $\mathcal{C}$.

Initially, $\mathcal{C}$ prepares its data values by augmenting it with (index) search keys. We abbreviate data values as *values* and the search keys as *keys* throughout this section. All other values and keys (e. g., cryptographic keys) are clearly differentiated if ambiguous. The values are stored in pseudo-random position. The keys are then inserted into a B^+-tree where the storage order of all nodes is pseudo-random, as well. The tree and values are linked by adding pointers to the leaves of the tree identifying the random position of the corresponding values. A value can be any data such as records in a relational database or files/documents in other database types. $\mathcal{C}$ then encrypts all nodes of the tree with a secret key SK_k and all values with SK_v. The encrypted B^+-tree and encrypted values are deployed on the untrusted cloud server D (see step ① in Figure 28[5]).

$\mathcal{C}$ uses SGX's RA feature for authenticating the enclave and establishing a secure connection between $\mathcal{C}$ and the enclave (we provide an explanation how to establish a secure channel to an SGX enclave in Section 2.3.2). Through this connection, $\mathcal{C}$ provisions SK_k into the enclave (step ②). This completes HardIDX's setup, which needs to be executed only once.

5 For visualization purposes, the tree nodes and values are shown to be encrypted as a block. In our HardIDX implementation each node and value is encrypted individually.

Now, $\mathcal{C}$ can send (index) search queries to D, which are encrypted with a probabilistic encryption scheme under SK_k. Hence, D and the untrusted $\mathcal{SP}$ do not learn anything about the content of a query, not even if the same query was sent before. When a query arrives in the enclave it is decrypted using SK_k (see step ③).

In step ④, the enclave loads the B^+-tree structure from the untrusted storage into enclave memory and decrypts it. Only the nodes of the B^+-tree are loaded, the values stored in the database are not loaded into the enclave. Given sufficient memory, the entire tree is loaded into the enclave and the search is performed afterwards (see step ⑤). As the tree size can exceed the memory available inside the enclave, we provide a second design. In this case, only a subset of tree nodes is loaded into the enclave. The tree is traversed starting from the root node and nodes are fetched from the untrusted storage when necessary. In both cases the search algorithm eventually reaches a set of leaf nodes, which hold pointers to values matching the query. This list of pointers, representing the search result, is output by the enclave to the untrusted server part (see step ⑥). To ensure that D, which is under control of the untrusted $\mathcal{SP}$, does not learn anything except the cardinality of the result set, the values in the result list are stored in a randomized order.

The result of the index search could be processed further, e. g., in combination with additional Structured Query Language (SQL) operators, in an SGX enclave in $\mathcal{SP}$ infrastructure. In order to complete the end-to-end secure search, we assume that D uses the pointers from the result set to fetch the encrypted values from untrusted storage and sends them to $\mathcal{C}$, where they are decrypted using SK_v (see step ⑦).

Notably, the plaintext values are never available on D. They are encrypted with strong standard cryptography methods (Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) in our implementation) and never decrypted on D, not even inside the SGX enclave. SK_v is only known to $\mathcal{C}$.

4.2.4 *Search Algorithms*

We designed two algorithms enabling secure range queries using SGX. As mentioned above we use an encrypted B^+-tree, which is processed inside an SGX enclave. For this approach we have to consider two cases: (1) The B^+-tree can be loaded into the enclave completely, or (2) the B^+-tree has to be loaded piecemeal into the enclave. We will briefly describe our two algorithm constructions below, a more detailed description is provided in our conference publication [149], which also includes proofs for the correctness and security of our second construction. The proofs for our first construction are provided in our journal article [151] and our technical report [150].

Our algorithms consist of two phases, (i) an initialization phase, and (ii) an operation phase. The initialization phase is the same for both algorithm constructions, while the operation phases differ in our two constructions.

4.2.4.1 *Initialization Phase*

In the initialization phase, the B^+-tree, which allows the efficient search for values and ranges of values stored in the database, is constructed. This construction is assumed to

be performed in a trusted environment, e. g., it can be done locally at the client side or in another SGX enclave. The B^+-tree structure and the values are encrypted with SK_k and SK_v, respectively, using an authenticated encryption scheme. The resulting encrypted data are transferred to D, where they are stored in untrusted memory. Furthermore, the HardIDX enclave is created on D and a secure channel is established between the initialization environment and the HardIDX enclave (see Section 2.3.2 for details on secure channel establishment via RA). Through this secure channel SK_k is provisioned to the HardIDX enclave.

4.2.4.2 *Operation Phase*

During operation, our two algorithm constructions work differently, we describe them below individually.

Construction 1. For the first case, our algorithm assumes that the entire B^+-tree is loaded into the HardIDX enclave. Either, the B^+-tree is small enough to fit in the physical memory available to the enclave, or SGX's demand paging functionality is used to load the required memory pages when needed.

First, the encrypted B^+-tree is loaded into the HardIDX enclave and all nodes are decrypted using SK_k, which was provisioned during the initialization phase. The B^+-tree can be loaded either precautionary or on-demand when a range query should be processed. Next, when a search query token τ is send by $\mathcal{C}$ to the server D, it is forwarded to the HardIDX enclave. The enclave decrypts τ and traverses the B^+-tree, resulting in a list of pointers to all values that fall within the range specified in τ. A random permutation of the list of pointers is output by the HardIDX enclave to D.D collects the encrypted values and sends them back to $\mathcal{C}$. Finally, $\mathcal{C}$ decrypts the values using SK_v.

Side-Channel Leakage. All de- and encryption operation of the enclave are performed using side-channel resilient implementations, in particular, Intel's hardware extension AES New Instructions (AES-NI) is used, which keeps all intermediate data of the cryptographic operation in registers, mitigating side-channel leakage and providing improved performance.

The initial decryption of the B^+-tree does not leak side-channel information about the structure of the tree. All nodes are decrypted sequentially, independent of the B^+-tree structure.

The traversal of the B^+-tree could leak the order of processed nodes. To prevent this, our algorithm ensures that for each node always *all* child nodes are accessed, whether they are actually processed or not. This way, the adversary $\mathcal{ADV}$ cannot learn which child node was actually relevant for a processed range query. However, $\mathcal{ADV}$ can observe the enclave's memory access patterns at page granularity, which enables $\mathcal{ADV}$ to learn relations between groups of nodes, where a group of nodes are all nodes that fit within a single page.

The final list of pointers to values in the result set is again randomized, thus $\mathcal{ADV}$ only learns the cardinality of the result set.[6]

6 Different methods could be integrated into HardIDX to hide result sizes [411].

Construction 2. Our second algorithm construction requires only a constant amount of memory for the HardIDX enclave. The nodes of the B^+-tree are loaded on-demand, i. e., only those nodes that need to be processed are loaded into the enclave.

As the first step, the B^+-tree's root node is loaded into the enclave and decrypted using SK_k. The second step is the same as in our first construction, i. e., when a search query token τ is send by $\mathcal{C}$ to the server D, it is forwarded to the HardIDX enclave, and decrypted in the enclave. Next, the range search in the B^+-tree is performed, where nodes that are not already loaded into the HardIDX enclave are requested on-demand from D's untrusted storage. To optimize communication overhead between enclave and untrusted host a list of nodes is created that need to be processed next and loaded into the enclave in a single batch operation. The batch size can be limited to a fixed size to achieve constant memory usage of the enclave. Eventually, all nodes of the B^+-tree, which are needed to identify values within the requested range, are loaded into the enclave, allowing the enclave to output the list of pointers to the encrypted values (again in a random permutation). The values contained in the result set are send to $\mathcal{C}$ that decrypts them using SK_v.

Side-Channel Leakage. Similar to our first algorithm construction, we use side-channel resilient implementation for all cryptographic operations (Intel's AES-NI).

In this construction, we do leak information about the B^+-tree structure, since $\mathcal{ADV}$ can observe which node are requested by the HardIDX enclave while the tree is traversed. However, the nodes of the B^+-tree are stored in a random order in D's untrusted storage, thus $\mathcal{ADV}$ only learns partial information about node relation, in particular, $\mathcal{ADV}$ learns which nodes were requested before others. In the best case $\mathcal{ADV}$ can conclude (incomplete) hierarchy information about the B^+-tree structure. The values or the order of leaf nodes is hidden from $\mathcal{ADV}$.

Similar to our first construction, the final list of pointers to values is randomized, thus $\mathcal{ADV}$ only learns the cardinality of the result set.[7]

4.2.5 *Performance Evaluation*

We evaluated HardIDX on an Intel Core i7-6700 processor at 3.40 GHz, with support for Intel SGX, and 32 GB DDR4 Random Access Memory (RAM). We used Ubuntu 14.04.1 (64 bit version) Operating System (OS) and the Intel SGX Software Developer Kit (SDK).

We tested different aspects of HardIDX, comparing the performance of our two algorithm constructions, in particular, the impact of the memory management for large search trees.

The details of our evaluation as well as a performance comparison to related work is provided in our conference publication [149], journal article [151] and technical report [150].

Performance Comparison. We measured the run time for our two algorithm constructions, with a tree branching factor of 10, i. e., each node in the B^+-tree has 10 child nodes, and 1 000 000 database entries, i. e., key-value pairs. The performance of our two

7 Different methods could be integrated into HardIDX to hide result sizes [411].

constructions converge for increasing numbers of values resulting from a range query. For small rages, with result sets $< 2^8$ elements, the second construction has a recognizable overhead, however, for larger ranges the difference becomes negligible.

Memory Management. To comprehend the effects of the different memory management approaches we evaluated our constructions with different tree sizes with up to 50 000 000 key-value pairs. Additionally, we tested four different branching factors (10, 25, 50 and 100). Our results show that a low branching factor of 10 has a significant impact on the first construction for tree sizes over 1 000 000 key-value pairs, while the second construction's run time remains constant for all tested tree sizes.

The sensitivity of our first construction lies in the increased memory page swapping that is required for large trees with low branching factors, as we elaborate in our conference publication [149].

4.2.6 *Conclusion*

HardIDX provides a highly performant solution to search for values and ranges of values over encrypted data. It utilizes SGX enclaves to process sensitive data in an untrusted environment, for instance the cloud. Our implementation minimizes the side-channel leakage of sensitive information about the database's content. This work shows that enclave-based systems can be hardened against side-channel attacks using engineering techniques, such as the use of side-channel resilient implementations of cryptographic primitives, combined with a careful design of the data processing patterns of used algorithms.

4.3 DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization

The main objective of Intel's Software Guard Extensions (SGX) is to protect the confidentiality and integrity of data processed in an enclave as well as the integrity of enclave execution. However, as has been repetitively demonstrated, SGX's isolation can be circumvented via software-based side-channel attacks [412, 389, 68, 340, 277, 174, 176]. A more detailed description of different side-channel attacks against SGX is provided in Section 4.4.1. Our own novel side-channel attack against SGX is presented in Section 4.1.

Oblivious RAM (ORAM) and Oblivious Execution provide general concepts to protect against information leakage at a very high cost, e. g., Obfuscuro [11] that implements oblivious execution for SGX induces 83 × overhead on average (up to 220 × in some cases). Other side-channel defenses for SGX rely on functionalities that are not available in all SGX-enabled Central Processing Units (CPUs) [353, 96, 182], while some approaches require manual effort and expertise from enclave developers, e. g., to annotate critical data [182, 323] or even redesign software [75]. Due to the disadvantages of existing side-channel defenses for SGX there is a demand for a more efficient, fully automated defense mechanism that relies solely on features available in *all* SGX-enabled CPUs.

Goals and Contributions. Our goal is to design a side-channel protection mechanism that prevents the adversary from learning the sensitive information processed inside an SGX enclave. Using fine-grained data randomization, we detain the adversary from inferring useful information from side-channel leakages. Our solution should work as an *automated* tool, which does not rely on enclave developers' assistance, and provide an *adjustable* security-performance trade-off.

The focus of this work is on information leakage caused by enclaves' data accesses, which was the target of many SGX side-channel attacks [68, 340, 277, 174]. Information leakage due to control-flow events in an enclave's code represents an orthogonal problem.

The main contributions of DR.SGX are as follows:

- We propose a novel side-channel defense approach for SGX. Our approach is based on a novel concept, called semantic-agnostic data randomization, which enables fine-granular data randomization without requiring information regarding the structure or semantics of the data.
- We present the design of DR.SGX, a fully automated tool implementing our semantic-agnostic data randomization scheme. DR.SGX permutes an enclave's entire data memory at cache-line granularity and instruments the enclave's code to operate on these permuted memory locations. Additionally, DR.SGX re-randomizes the data memory repeatedly at a configurable frequency.
- We evaluate DR.SGX's performance and analyze possible leakages. We show that the leakage of DR.SGX depends on properties of the target enclave, i. e., whether

an enclave executes memory accesses following patterns that are predictable to by adversary.

4.3.1 *Model and Assumptions*

In this work, we focus on systems that provide an isolated execution environment, which is implemented as an execution mode of the main CPU. In particular, the CPU's shared resources, such as caches, are used by all execution modes of the CPU and thus are shared between isolation domains. Our work is targeted towards Intel SGX, however, the same model also applies to other architectures such as ARM TrustZone [24] or software-based isolation solutions [266].

4.3.1.1 *Problem Space*

Side-channel attacks on software in general, and SGX in particular, come in many different forms. Any kind of resource use that is influenced by the software's execution and can be observed by an adversary, can serve as a side channel. For instance, the use of electricity as well as effects thereof, such as electro-magnetic emission, or the use of shared CPU caches. In this work we focus on *software* side channels, i. e., those that are observable by a software program running on the target machine, precluding physical or hardware side-channel attacks.

In the realm of software side-channel attacks, a number of distinct variants exist. On one hand, different shared resources can be used as a side channel, for instance the different caches of the CPU, or the virtual memory management. On the other hand, side-channel attacks can target different information, including sensitive access patterns to data as well as secret dependent code execution paths.

In this work we focus on software side-channel attacks that target *data accesses*. We consider attacks that aim at inferring the control flow of a program an orthogonal problem. Our rationale is two-fold. First, many side-channel attacks on SGX have been based on data access patterns [68, 340, 277, 174]. Furthermore, our solution can be combined with existing protections against control-flow leakage attacks, for example with the Zigzagger approach proposed by Lee et al. [239].

4.3.1.2 *Adversary Model*

The adversary's goal is to extract sensitive information from an isolated execution environment (enclave) through cache side-channel attacks [68, 277, 340, 119, 184, 405] (including CPU-internal caches such as the Translation Look-aside Buffer (TLB) [175, 176]) and/or paging side-channel attack [412, 389]. Sensitive data in this context are not limited to cryptographic keys, which are the "classical" targets of side-channel attacks. Instead, sensitive data have to be seen much broader, for instance, when processing privacy-sensitive data in the cloud [68].

The adversary can freely configure and modify all software of the system, including privileged software such as the Operating System (OS). The adversary knows the initial

memory layout of the target enclave, i. e., the code and initial data of the enclave. Furthermore, we assume that the adversary can initiate the enclave as often as desired.

However, the adversary cannot directly access the memory of the enclave. The internal processor state (e. g., the CPU registers) is inaccessible to the adversary, in the event of an interrupt the state is securely stored in an isolated memory region. The adversary cannot modify the code or initial data of the enclave, as the enclave's integrity can be verified using Remote Attestation (RA).

Attacks such as Meltdown [247] and Spectre [225], which circumvent isolation boundaries using transient execution, constitute an orthogonal problem. While these attacks can be applied to SGX [95, 390], they can be prevented by other means. Intel provides security updates for SGX that prevent those attacks [95]. RA allows a remote party to verify that these updates are used before provisioning secret information to an enclave. The more general problem of data-access driven side-channels is much harder to solve in architectures such as SGX. DR.SGX addresses the latter and more difficult problem.

We assume the position of the adversary to be as strong as possible and therefore we assume the adversary to have a noise-free cache side-channel and to be able to obtain a "perfect memory access trace" of an enclave. This means that the adversary can observe all memory accesses of an enclave, e. g., using a cache attack technique such as Prime+Probe [294]. The adversary can precisely determine which cache line has been used by an enclave and also the order in which the cache lines have been accessed. The adversary cannot extract information which is more fine grained than accesses to cache lines, i. e., the offset inside a cache line is not observable to the adversary (see Section 4.4.1 for a discussion of possible attacks with finer granularity). Additionally, for each memory access, the adversary can gain information about the accessed memory pages of an enclave [412, 389].

More formally, trace $t = \{c_1, p_1\}, ..., \{c_n, p_n\}$ is an ordered list of side-channel observation pairs that capture every memory access that a victim enclave makes. In each observation pair, c_i is the part of the memory address that determines the cache line, the accessed address gets mapped to and p_i is the part of the address that determines the accessed memory page. On current Intel CPUs the cache line size is $64\,\mathrm{Byte}$, thus, the last six bits of an address are oblivious to the adversary.

4.3.1.3 *Design Goals*

General statements about which memory accesses of a program could leak sensitive information are hard to make in practice. All memory accesses must be assumed to potentially leak information if the adversary can associate them with relevant data elements or structures. For the adversary it is sufficient to distinguish two memory locations to learn one bit of information. Those memory locations could be two different data structures, e. g., two variables, or different elements within the same data structure, e. g., different entries in a table. To protect all possible programs, the data structures of a program as well as the elements within data structures need to be randomized.

The goal of our work is to provide a protection mechanism against side-channel attacks that can be applied to *arbitrary enclave programs without developer assistance*. In

particular, the developer must not be required to follow any rules or guidelines for programming applications or add annotations to the source code. While annotating "critical" data in general helps improving the performance of most solutions, it is also error-prone: especially in non-cryptographic applications, it is not always obvious which accesses to data objects might leak sensitive information. This is crucial as most software developers are not security experts and cannot comprehensively identify data that could leak information.

The goal of DR.SGX is to provide a trade-off between security and cost in the design space that extends from unprotected processes over to SGX enclaves without protection against side-channel leakage, and to enclaves with DR.SGX to ORAM solutions. By default, SGX enclaves impose low performance overhead in order to process data in isolation while not protecting against side-channel leakage. ORAM can provide comprehensive protect against side-channel attacks, when implemented correctly. However, schemes implementing ORAM for all memory access of an enclave, such as Obfuscuro [11], cause very high performance overhead (Obfuscuro reports an average overhead of 83 × and a maximum overhead of up to 220 ×). DR.SGX provides a flexible and adjustable solution, which allows to find a trade-off between security and performance. In the extreme case, i. e., when re-randomizing the enclave memory after every memory access, DR.SGX effectively implements ORAM. However, in the best case, DR.SGX's overhead is less than 5 × while providing protection for enclaves without predictable accesses to sensitive data structure (see Section 4.3.5 for a detailed evaluation of DR.SGX's security properties). For most enclaves, a configuration between those two extremes can be chosen. A synopsis of our evaluation results for different configuration is discussed in Section 4.3.4, detailed evaluation results are provided in our conference publication [72] as well as our technical report [66].

4.3.2 *DR.SGX Concept and Design*

The core idea of our DR.SGX concept is to break the link between side-channel observations made by an adversary and the sensitive information processed by the victim. Side-channel attacks inherently rely on the correlation between an observable effect and the data the adversary aims to extract. Our defense obfuscates the link between memory locations and data elements. Data elements are located at randomized memory locations, hence, the adversary cannot deduce which data element was accessed from an observed memory access location. The adversary no longer learns *which* data element was accessed; the adversary only learns that *some* data element was accessed.

DR.SGX splits enclave memory into small blocks that are randomly reordered, resulting in an unpredictable memory layout from the adversary's perspective. Figure 29 illustrates the concept on the example of an S-box from an Advanced Encryption Standard (AES) implementation. By default, the S-box (`FSb`) is stored as an array in consecutive memory at a predictable location, shown on the left as initial memory layout L_0 in Figure 29. Through a cache side channel an adversary can observe which part of the S-box is accessed. Since the accesses to the S-box depend on the secret key, the adversary can use this information to recover the key. However, the adversary cannot observe accesses to individual bytes

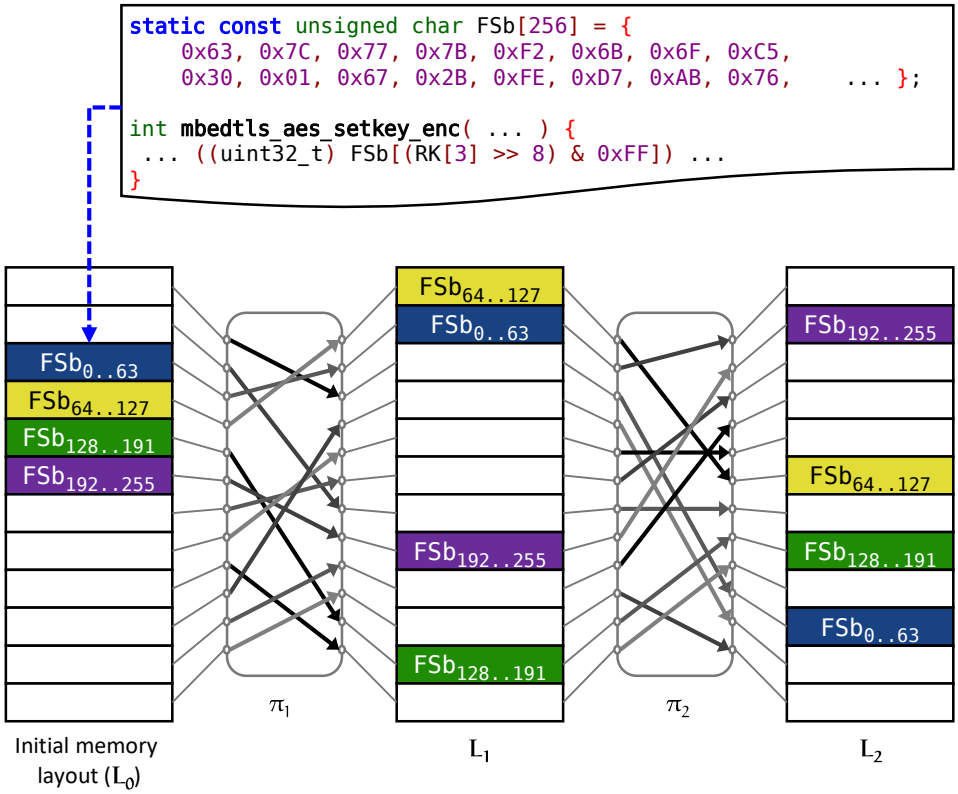


Figure 29: DR.SGX's memory block randomization splits large memory structures such as arrays into small blocks and reorders them. During the run time of an enclave its memory layout is re-randomized using the permutation function π. Each memory block is the size of a cache line (64 Byte), i. e., the finest granularity observable by the adversary.

of the S-box. The adversary can only make observations at the granularity of cache lines (64 Byte). DR.SGX divides *all* data memory of an enclave into blocks of cache line size, illustrated by the blocks forming L_0 in Figure 29. These blocks are reordered by a permutation function π_1, resulting in a randomized memory layout L_1. Throughout the run time of an enclave, the memory layout is constantly re-randomized: by applying a permutation function π_2 on L_1 a new and different memory layout L_2 is created. As a result, the memory locations and thus the cache lines corresponding to the S-box are frequently changing, hindering the adversary's ability to link observed (cache or page) accesses to the S-box.

4.3.2.1 *Requirements and Challenges*

Below, we describe the main challenges we had to tackle in order to implement our idea of semantic-agnostic data randomization.

Semantic Gap. Providing side-channel protection through data randomization without developer assistance (e. g., code annotations) is a challenging task due to the semantic gap that is inherent to memory-unsafe programming languages such as C and C++.

Currently, C and C++ are the only programming languages officially supported in the Software Developer Kit (SDK) provided by Intel for the development of SGX enclaves.

Re-randomization. Randomizing the memory layout of a program once to prevent an adversary from learning which data are accessed is not sufficient. The adversary can determine the relation of memory locations and data objects based on various information. For instance, the initialization of data structures can reveal data locations. In the example in Figure 29, the S-box is initialized during the creation of the enclave, however, other AES implementations initialize the S-box at run time which allows the adversary to learn the locations of all parts of the S-box array *after* the initial randomization of the memory layout. Similarly, access frequency can reveal the randomized location of data elements: if a particular object is accessed a predictable number of times the adversary can identify the object by finding the memory location that was accessed the expected numbers of times (frequency analysis). To thwart the adversary in recovering the randomized memory location of data objects, their locations need to be changed throughout the run time, such that the adversary cannot link data accesses to data objects.

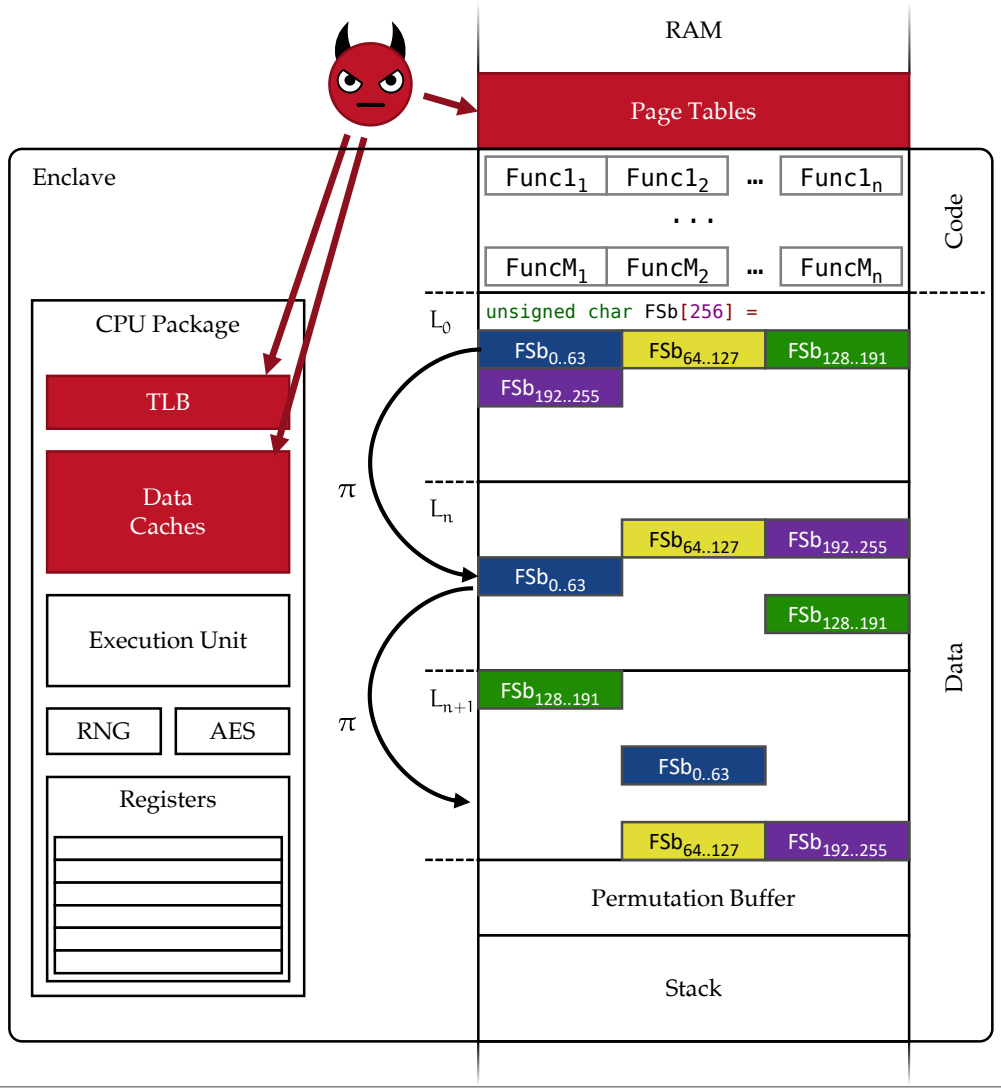
(Re-)randomization under Adversary's Observation.
All memory-related actions of the attacked enclave can be observed by the adversary, including those required during initial data randomization and during the re-randomization of the memory layout. The initial (un-randomized) memory layout is known to the adversary, i. e., the adversary can monitor memory events while data is copied to its randomized locations. Similarly, if the adversary managed to recover information about the randomized memory layout L_n the adversary could link the re-randomization operations used to transfer data from L_n to L_{n+1}, and thus, also gain knowledge about the new layout L_{n+1}. Therefore, the randomization has to be done in such a way that its effects are not observable by the adversary.

4.3.2.2 *DR.SGX Design*

Our solution, a compiler-based tool called DR.SGX, addresses the design goals and challenges described above by randomizing *all* enclave data at fine granularity and re-randomizing the data continuously throughout the run time of the enclave.

Figure 30 shows the system view of DR.SGX. The Trusted Computing Base (TCB) of an SGX enclave includes the CPU package and an isolated section of the Random Access Memory (RAM). However, the CPU caches, TLB and the page tables are observable by the adversary. The data cache of the CPU can be used to observe memory access patterns of an enclave. On the other hand, the paging mechanism can be exploited in different ways to learn about memory reads and writes by an enclave, e. g., by observing cache conflicts in the TLB, the adversary learns which memory pages were used. Additionally, the adversary has control over the page tables allowing the adversary to learn which memory pages an enclave accessed.

However, an SGX enclave also includes components that cannot be attacked through software side channels. The CPU's registers and accesses to them cannot be observed by



AES: Advanced Encryption Standard
RNG: Random Number Generator
TLB: Translation Look-aside Buffer

Figure 30: DR.SGX system design. The main memory of an enclave is not directly accessible by the adversary; however, the adversary can observe memory access indirectly through cache and paging side channels. The CPU's internal state stored in registers and/or special function units (e. g., the AES engine) are not observable by the adversary.

the adversary.[8] Also, the execution unit and special function units, such as the Random Number Generator (RNG) or the AES engine, are secure when operating over registers. DR.SGX combines these parts and function units of SGX that are secure against side channels, to obfuscate main memory accesses to the adversary.

8 The LazyFP [363] attack cannot be used on SGX enclaves since the register state is cleaned by the processor before exiting an enclave.

DR.SGX performs randomization at granularity of cache lines, the finest granularity at which the adversary can distinguish memory accesses (Section 4.3.1). Figure 30 shows how DR.SGX uses a random permutation function π to reorder the program's data in memory. Since the adversary cannot identify individual elements within a single cache line, accesses to the first array element (`FSb[0]`) and the 64th element (`FSb[63]`) are indistinguishable for the adversary. The randomization is based on secret values which are generated and only accessible *inside* the enclave. They are only processed by the hardware AES engine of the CPU. The CPU's AES engine holds all state and intermediate results in registers, which are not observable by the adversary, hence, the adversary cannot learn about π through cache or paging side channels.

DR.SGX randomizes global variables and the heap. The stack cannot be easily randomized since the hardware expects it to be contiguous. Thus, variables on the stack larger than a cache line are moved to the heap and replaced by a pointer on the stack. The remaining variables are protected using multiple different memory layouts: for every function n variants are created ($\texttt{Func1}_1$, $\texttt{Func1}_2$, ..., $\texttt{FuncM}_n$ in Figure 30), all with different stack memory layouts. This approach is similar to the one proposed by Crane et al. [113]. The parameter n can be chosen by the enclave developer, Crane et al. [113] suggest $n = 10$ different variants. On every invocation of a function, one of its n variants is chosen randomly.

The size of the memory region (heap) for the enclave's data is a parameter of the permutation function π (see Section 4.3.3).

Memory Access Instrumentation. DR.SGX performs randomization on cache line granularity for two reasons: (a) randomizing at finer granularity provides no security advantages, and (b) randomizing in a data-structures-aware fashion is impractical due to the semantic gap. Our randomization requires that *all* memory accesses are instrumented, which we ensure using an automated compiler pass. The program code determines the memory location (i. e., address) of the data in the original, un-randomized layout. Then, before the access is performed, the randomized location of that address is calculated, using the permutation function π. The data is then accessed in its new, randomized location.

As we will elaborate in later sections, the cost of performing the randomization calculation for *every* memory access is significant. We overcome this problem by implementing a "permutation buffer". The permutation buffer, similar to an address translation cache, holds the randomized locations of recently used data. Hence, for data locations stored in the permutation buffer, the function π does not need to be recalculated. However, accesses to the permutation buffer itself must be protected from leaking information. Therefore, the buffer is accessed in an oblivious way.

Initial Randomization. The initial randomization of the enclave's data needs to be done in a way that cannot be observed by the adversary, to keep the adversary from learning the randomization function π or the new memory layout. In particular, if the adversary can observe a read operation from the un-randomized initial memory layout and a subsequent write operation to a randomized address, the adversary can link data structures to the randomized memory locations.

A general approach to break this linkage is to load a set of data into CPU registers (register operations cannot be tracked by the adversary) and write the data in a random order to their new locations. This approach, however, is limited in the amount of the data that can be loaded into registers at once, enabling the adversary to learn partial information about the randomized memory layout.

DR.SGX uses a randomization method which hides fine-grained (cache-line granularity) memory locations from the adversary. Specifically, we use *non-temporal writes* [208] that evade the CPU's caches; therefore, the adversary cannot observe which memory addresses are written during the initial randomization. Although the non-temporal writes prevent accesses to the new memory layout L_1 from being cached, the adversary can still observe the written memory locations through the more coarse-grained paging side-channel (i. e., the adversary's trace contains a page event p_i and no cache event c_i for the non-temporal write). This allows the adversary to know, for each memory block read from the previous memory layout L_0, to which memory page it was written in L_1. However, multiple cache lines are written to each page: assuming 4 kB pages, 64 cache-line-sized memory blocks will be written to the same page. To hide this access pattern the initial randomization of DR.SGX accesses *all* memory pages of L_1 for each memory block that is moved, see Section 4.3.5.

DR.SGX continuously re-randomizes the memory layout. Starting from the initial memory layout L_0 a random permutation function π_1 is applied to derive the first randomized layout $L_1 = \pi_1(L_0)$. After a configurable window w the memory layout is re-randomized, applying π_2 to derive $L_2 = \pi_2(L_1)$.

Similar to the initial randomization, the adversary (by observing reads from L_n and writes to L_{n+1}) could link those two operations to learn the relation between those memory layouts. Again, DR.SGX uses non-temporal writes to hide this information. In Section 4.3.5 we explain how a small number of re-randomization rounds hides the location of the element from the adversary completely.

4.3.3 *DR.SGX Implementation*

The key components of our DR.SGX implementation comprise (i) the instrumentation of memory access operations, (ii) our permutation function, (iii) the initial randomization procedure, (iv) our stack data randomization implementation, (v) the permutation buffer, and (vi) our re-randomization procedure.

All these components are integrated into an enclave using automated compiler-based instrumentation. Our implementation of DR.SGX extends the LLVM compiler [256] to instrument an enclave's code in the Intermediate Representation (IR). Furthermore, additional function blocks (e. g., the permutation function) are inserted into an enclave in the compilation process.

We will explain each of DR.SGX's key components shortly below. The details of our implementation are provided in our DR.SGX conference publication [72] and our technical report [66].

Memory Access Instrumentation. DR.SGX transforms the linear (virtual) memory layout of an SGX enclave into a randomized layout. Therefore, all memory accesses need to be translated to access data at their new randomized memory locations rather than the original memory addresses in the linear memory layout. Our implementation instruments all memory access operations to calculate from the linear memory addresses the randomized locations based on the permutation function π. By instrumenting memory operations at the IR in the LLVM compiler DR.SGX is agnostic to the accessed data structures.

Random Permutation. DR.SGX's randomization function needs to fulfill two requirements. It must be (1) collision free, i. e., the randomized location of two different data elements must be different. And (2) the randomization must be based on a non-secret algorithm. It should be only dependent on a small secret that can be generated inside the enclave and store in a register at all time.

Our DR.SGX implementation is based on the FFX Format-Preserving Encryption scheme [45]. It allows to generate random permutations of the memory layout (fulfilling the first requirement), using a symmetric cryptographic algorithm (fulfilling the second requirement). We utilize AES New Instructions (AES-NI), benefiting from its performance as well as its side-channel resilience.

Initial Randomization. DR.SGX's initial randomization, utilizing DR.SGX's random permutation functionality, is crucial since an enclave's initial memory layout L_0 is known to the adversary. During initial randomization, the memory is copied in blocks b, each the size of one cache line, from L_0 to the first randomized layout L_1. To prevent leaking information about L_1 the transfer must be implemented in a side-channel resilient way. The adversary must not learn the position of any block in the new memory layout L_1. Blocks are written to L_1 using non-temporal write operations. Non-temporal write operations bypass the CPU's caches, hence, causing no effect in the cache observable by the adversary. To prevent leakage through the page-based memory management, DR.SGX accesses all memory pages of L_1 for every block written to L_1.

Stack Randomization. Our DR.SGX implementation moves large data objects, i. e., objects lager than a single cache line, from the stack to the heap where they are subject to DR.SGX's fine-grained (re-)randomization. All remaining data objects are randomly reordered on the stack, similar to the code randomization methods proposed by Crane et al. [113]. In particular, for every function n versions with different stack layouts are created at compile time. During run time, one of these n versions is selected randomly whenever a function is called.

Permutation Buffer. DR.SGX randomizes memory in blocks b that are the size of a cache line, i. e., the finest granularity observable by the adversary. Hence, for all memory accesses to the same block b the permutation function must be calculated only once. To improve performance, in particular of sequential memory accesses, our implementation buffers these calculations in a *permutation buffer*. However, the usage of the permutation buffer itself might leak sensitive information, such as secret dependent access patterns. Therefore, all accesses to the permutation buffer are implemented in an oblivious way.

Re-randomization. DR.SGX frequently re-randomizes an enclave's memory layout. Similar to the initial randomization the old layout L_n is copied to a new memory layout L_{n+1}

in blocks of cache line size. Again, non-temporal write operations are used to prevent leakage through cache side channels. However, in favor of performance, leakage through the page-based memory management is not prevented. We analyze the security of this implementation choice in Section 4.3.5.

4.3.4 *Performance Evaluation*

We evaluated the memory overhead as well as the run-time overhead of DR.SGX on an Intel i7-6700 processor (3.40 GHz) with 128 MB Enclave Page Cache (EPC), running Ubuntu 14.04.4. A detailed evaluation is provided in our conference publication [72] and our technical report [66].

Memory Overhead. DR.SGX provides protection for data stored on the heap as well as the stack. Heap randomization requires the enclave to allocate two times the heap size (only) during re-randomizing, i. e., when coping all data from the current memory layout L_n to the next layout L_{n+1}. DR.SGX's stack randomization introduces n version of every function, increasing the enclave's code size by a factor of n.

Run-time Overhead. We evaluated DR.SGX's run-time overhead using the benchmark suite Nbench [79], which we adapted to work in an SGX enclave.

DR.SGX's overhead is dependent on different configurable factors. The heap size h directly impacts the cost of the re-randomization operations, which copies the entire heap from the current memory layout L_n to the new layout L_{n+1}. The second factor is the frequency f at which re-randomization is performed, i. e., more frequent re-randomization leads to larger performance overhead.

For heap sizes $h = 4$ MB, and re-randomization frequencies f between once every 300000 memory access operations and once every 10000000 memory access operations, DR.SGX's geometric mean overhead ranges from 5.45 × to 12.21 ×.

Without re-randomization, i. e., when applying only the initial randomization and stack randomization, the geometric mean of DR.SGX's run-time overhead is 4.36 ×.

4.3.5 *Security Analysis*

Subsequently we analyze the security of DR.SGX's protection of data on the heap, which uses our novel semantic-agnostic randomization approach. Our stack protection follows a previously evaluated approach [113], therefore we do not provide an explicit analysis of its security properties. However, it is important to note that DR.SGX replaces large stack allocations with heap allocations, i. e., all large data structures process in an enclave protected with DR.SGX benefit from the frequent re-randomization applied to the heap memory.

4.3.5.1 *Assumptions and Model*

As detailed in our adversary model (cf. Section 4.3.1.2) the goal of the adversary is to extract secret data from an enclave protected with DR.SGX by observing memory

accesses patterns through one or multiple side channels. In particular, we assume that the adversary can extract a perfect trace of all cache events and page faults. This over-approximates the adversary's capabilities, in practice all known attacks exhibit significant noise in the cache channel [68, 340, 277, 174].

DR.SGX is focused on data-driven side-channel attacks, which we model as follows. The target enclave contains secret data s of arbitrary length, e. g., cryptographic keys, medical data, or intellectual property. The enclave contains a data structure d that is processed depending on s. In particular, d consist of n elements $e_1 \dots e_n$ that are accesses during the processing of s, the access pattern to the elements e_i is dependent on s. Example of such data structures include look-up tables, e. g., S-boxes as used in cryptographic algorithms such as AES, hash tables, or in-memory databases. In our model, the elements e_i of d are of the size of a cache line, i. e., the finest granularity at which the adversary can observer access patterns. Hence, smaller elements of d are indistinguishable for the adversary. Logical elements of d that are larger than a single e_i can be observer by the adversary by combining all observation for $e_i \dots e_{i+k}$ elements that fall into the logical element (assuming the logical element is k times the size of a cache line and aligned).

The target enclave makes l access to potentially different elements of d, depending on s. The adversary aims to learn this access pattern. However, not all accesses to d and its elements $e_1 \dots e_n$ must be dependent on s. Predictable accesses to d, e. g., during initialization, can be filtered out by the adversary.

Attack Trace Positioning. We assume the adversary can identify the "attack position" in the side-channel trace, i. e., the position in the trace that corresponds to secrete dependent data accesses during the enclave's execution. The adversary can analyze the enclave, for instance, by executing it without DR.SGX's protection or by inspecting its source code, to learn at which point secret dependent accesses take place. When executing the protected enclave, the adversary can count the number of accesses of the enclave to be expected before the secret dependent access occurs,[9] assuming deterministic enclave's execution.

4.3.5.2 *Inferring Secret Enclave Data*

We distinguish two types of enclaves for our analysis, which we discuss separately: (1) enclaves that do not have predictable accesses to the data structure d, and (2) enclaves implementing a predictable access to d (e. g., an initialization of d).

Enclaves without Predictable Accesses. DR.SGX provides strong protection for enclaves *without* accesses to a data structure d that are predictable for the adversary, i. e., not secret dependent. The initial randomization performed by DR.SGX is done in such a way that the adversary cannot learn the randomized layout of d. In particular, from the initial memory layout L_0 that is known to the adversary DR.SGX transfers all data (including d) to a new randomize layout L_1. This is done in blocks of cache line size, using non-temporal write operations. The read operations from L_0 are observable by the adversary as cache events, however, they do not reveal any secret information as

9 Compared to the number of memory accesses observed by the unprotected enclave, the adversary has to consider the additional memory access of DR.SGX.

all memory of L_0 is accessed sequentially. The non-temporal write operations to the new randomized location L_1 do not lead to cache events observable by the adversary. To prevent leakage through a page fault side channel, DR.SGX sequentially accesses *all* memory pages of L_1 for every written block in a fixed sequential order, i. e., the adversary does not learn to which page in L_1 a block is written. Since the adversary cannot observe the positions to which any block is copied in L_1, the memory layout of L_1 is completely unknown to the adversary. Since no predicable accesses to d occur, the adversary cannot learn the randomized layout of d, and thus, induce no information for observed memory accesses.

Enclaves with Predictable Accesses. Enclaves that perform predictable accesses to the data structure d, e. g., initialize d by setting all elements $e_1 \dots e_n$ to an initial value, will reveal the randomized layout L_p at the time the predictable access is performed. More precisely, the permuted memory location of each accessed e_i is learned by the adversary.

The following permutation rounds $L_{p+1}, L_{p+2}, \dots$ of DR.SGX will re-permute the elements location. The adversary traces the relocation of all elements e_i to learn the memory layout L_s when the secret dependent access takes place.[10]

During the re-randomization, DR.SGX reads blocks b (each the size of a cache line) sequentially from the old layout L_n and writes them to the new layout L_{n+1}. When writing a block b to L_{n+1}, non-temporal writes are used, hence, no cache events result from the write operation. The adversary can only learn from the page-fault side channel which memory page a block was copied to. Within this page, a block b can be placed at $4\,\mathrm{kB}/64\,\mathrm{Byte} = 64$ different offsets. Hence, the adversary's certainty regarding the location of a known block b in L_n being 1 is reduced to $\frac{1}{64}$ in L_{n+1}.

With each subsequent re-randomization of DR.SGX, the adversary's uncertainty increases rapidly. Given a limited heap size, the probability that a block b can be in every memory page with equal likeliness rises quickly. We show in our conference publication that with high probability a block's location is equally likely in any memory page after four re-randomization rounds for a 2 MB heap [72].

Hence, the security of DR.SGX depends on the number of re-randomization rounds performed between the predictable access to d that is observable by the adversary and the secret dependent access to d.

4.3.6 *Conclusion*

In this section we introduced semantic-agnostic data randomization as a new defensive approach against side-channel attacks on SGX. Our design and implementation of DR.SGX allows the instrumentation of enclave code such that all data locations in enclave memory are permuted at cache-line granularity and re-randomized at run time. Unlike previous defenses, our solution allows non-expert developers to harden their enclaves against various data-driven attack strategies with an adjustable security-performance trade-off.

10 The adversary does not need to learn the entire memory layout L_s, knowing the position of d's elements $e_1 \dots e_n$ (or a subset of them) is suffices.

4.4 Related Work

In this section, we review works related to both, our side-channel attack (Section 4.1) and our two defenses (Section 4.2 and Section 4.3). First, in Section 4.4.1, we discuss side-channel attacks and compare them to our own attack and explain how they relate to our defense DR.SGX. Afterwards, in Section 4.4.2, we explain different defense strategies against side-channel attacks and discuss their effectiveness against our attack and compare them with our own defense DR.SGX.

HardIDX is not focused on defeating side-channel attacks targeting enclave's Random Access Memory (RAM) accesses, its main objective is to use enclave-external storage in an oblivious way for databases. We discuss alternative approaches for scenarios where a database is outsourced, e. g., to the cloud, in Section 4.4.2. More general approaches to secure applications using Trusted Execution Environments (TEEs) and different concrete applications are discussed in Section 5.3.

4.4.1 *Side-Channel Attacks*

In the past, many side-channel attacks have been developed targeting various systems and platforms. In this section, we focus on side-channel attacks that target TEEs, specifically Intel Software Guard Extensions (SGX), and that are closes to the settings considered in this chapter. We cover both, cache-based side channels leaking information from SGX enclaves as well as attacks exploiting leakage through the dynamic memory management of SGX. While the latter type only emerged due to the adversary model of SGX, i. e., an untrusted Operating System (OS) is responsible for managing an enclave's memory, the basic concepts of cache-based side-channel attacks have been developed before SGX was introduced; we first discuss selected cache-based side-channel attacks not targeting SGX enclaves.

Cache-based Side-Channel Attacks. The first cache-based side-channel attack [308] demonstrated information leakage via L1 cache. It was successfully applied to reveal Rivest–Shamir–Adleman (RSA) keys of OpenSSL implementation through monitoring accesses to a table with precomputed multipliers, which are used by the algorithm throughout the exponentiation.

Osvik et al. [294] formalized two cache-based side-channel attack techniques, *Evict+Time* and *Prime+Probe*, which have been used to attack various cryptographic implementations [284, 381], have been applied to Last Level Cache (LLC) and have been used to build cross-core side channels [210, 250]. Furthermore, these attack techniques were shown to be applicable to mobile and embedded platforms [54, 407, 361, 360]. In the context of cross-core attacks, new and more complex attack techniques were developed, such as Flush+Reload [414], Evict+Reload [180], and Flush+Flush [181]. These attacks typically target cryptographic libraries and require tens of thousands of repetitions, while our attack concentrates on non-cryptographic applications and our attack technique requires much fewer execution traces.

Uhsadel et al. [387] study the use of Hardware Performance Counters (HPCs) for side-channel attacks. They use HPCs to observe the behavior of their victim directly, e. g., record cache hit/miss events of the victim. This approach is not suitable for SGX enclaves because enclaves do not update HPCs due to Intel's Anti Side-Channel Interference (ASCI). In contrast, we use HPCs (called Performance-Monitoring Counters (PMCs) in Intel CPUs) to record cache events of the adversary's Prime+Probe code.

Fine-grained Cache Leakage. Yarom et al. [415], Moghimi et al. [278] have investigated the possibility of leaking information through side channels with granularity smaller than one cache line.

According to Intel, accesses within the same cache line with different offsets may have deviating access times [205]. This is due to cache-bank conflicts, which occur in case of concurrent accesses to different cache banks, resulting in some of the conflicting requests being delayed. This was exploited by CacheBleed [415], an attack that successfully recovered 60 % of an RSA key's exponent bits (which is sufficient to recover the remaining bits efficiently by other means [185]) after observing 16 000 decryptions. We investigated applicability of the CacheBleed attack to SGX enclaves and confirmed that the CacheBleed attack is not applicable to processors with SGX support due to updates in cache architecture of CPUs supporting SGX.

MemJam [278] uses read-after-write false dependencies to introduce latency when a victim program reads data with a specific page offset. By measuring the run time of the victim program, a high number of times, while *jamming* different page offsets, the adversary can infer which offsets are read more often by the victim. This attack can leak information with a four-byte granularity but requires an extremely high number of runs (50 *million runs* for an attack against a simple and deterministic SGX enclave). However, with DR.SGX, the page offsets of data change between different runs, making the correlation of timing information for different runs exponentially more involved. Moreover, the accesses due to DR.SGX's own code generate a significant amount of noise. Finally, the code of DR.SGX itself was designed to not be vulnerable to MemJam attacks, e. g., by randomizing the permutation buffer layout (details of our permutation buffer randomization are provided in our conference publication [72]).

SGX Side-Channel Attacks. Since SGX's introduction, it has been hypothesized that side-channel attacks could be mounted against SGX enclaves, e. g., by Costan and Devadas [109]. Xu et al. [412] demonstrated page-fault side-channel attacks on SGX, where an untrusted OS extracts secrete information from protected applications by tracking memory accesses at the granularity of memory pages. Van Bulck et al. [389] advance the memory-paging-based side-channel attack by Xu et al. [412] by changing the way an untrusted OS observes an enclave's usage of memory page to circumvent defenses, which are based on the premise that an attack will cause exceptions (i. e., page faults) as a side-effect [353, 354]. They use of memory page attributes, in particular the "dirty" and the "accessed" bits, and monitoring of the caching behavior of an enclave's page tables to learn which pages were used by an enclave. DR.SGX can thwart paging-based side-channel attacks even if they do not interrupt the enclave execution since DR.SGX does not aim to prevent the adversary from making observations (or detect an ongoing

attack), DR.SGX rather prevents the adversary from gaining (useful) information from observations the adversary can make.

Gyselinck et al. [183] show that the x86 segmentation unit can be used to leak information about an enclave's execution, as well. It allows an adversary to monitor memory accesses at page granularity, similar to previous attacks [354], without changing the page tables. DR.SGX's defense is agnostic to the monitoring method used by the adversary; hence, it provides the same protection against these attacks as for previous attacks with page-granularity leakage. Additionally, for the first one MB of enclave memory, this attack allows to extract information, in particular control-flow information as well as instruction size, at the granularity of single bytes. However, this fine-grained information leakage was addressed by a microcode update from Intel.

TLBleed monitors the Translation Look-aside Buffer (TLB) to learn memory accesses of enclaves [176]. While this approach, similar to our attack, does not require the frequent interruption of the attacked enclave, it reveals only course-grain, i. e., page-granularity, information. DR.SGX thwarts this attack as it randomizes the entire enclave memory, thus the adversary only learns the access patterns to random memory locations, regardless of the side-channel used to monitor memory accesses.

Lee et al. [239] use branch shadowing, i. e., indirectly monitoring the Branch Target Buffer (BTB), to infer the control flow of an enclave. Similarly, BranchScope uses the directional branch predictor to infer the control flow of enclaves [143]. These approaches require the victim enclave to be interrupted at a high frequency, which enables effective detection methods discussed below [353, 96] (cf. Section 4.4.2).

Cache-based SGX Side-Channel Attacks. Multiple cache-based side-channel attacks targeting SGX enclaves have been developed in parallel to our attack (presented in Section 4.1). Below we elaborate on the difference between these attacks and ours.

Schwarz et al. [340] study a scenario, where an unprivileged attacker process (hiding in an enclave) is spying on the L3-cache utilization of another process (or enclave). The main difference to our work is that their attack monitors L3 (cross Central Processing Unit (CPU) core), while our attack works on L1 (on the same CPU core), i. e., our attack techniques are largely different.

Moghimi et al. [277], Götzfried et al. [174] target Advanced Encryption Standard (AES), and isolate the victim and attacker code on a single CPU core, such that they share the L1 cache.

CacheZoom [277] attacks an AES implementation through L1 cache by interrupting the victim, and thus increasing the temporal resolution of the attack. Enclave exits introduce noise in a subset of cache lines rendering them unobservable. Additionally, the interrupts make the attack easily detectable [353, 96]. Dall et al. [119] use a side-channel attack such as CacheZoom, i. e., a high-resolution Prime+Probe attack, to extract partial key information from SGX quoting enclave.

The attack by Götzfried et al. [174] aims at extracting the secrete encryption key from an AES implementation in an SGX enclave. The attack leaks information through the L1 cache, i. e., requiring the victim and attack code to execute on the same CPU core. Similar to our attack, they run the victim uninterrupted to avoid disturbance due to enclave exits. However, their attack assumes synchronization (collaboration) between the

victim and the adversary – an assumption which typically does not hold in practice. In particular, they assume that (1) the victim and attacker code run as a single process in two separate threads; (2) victim and attacker code have a shared memory, which they used to communicate and exchange data, e. g., the adversary provides cipher texts that need to be decrypted by the victim; (3) the victim synchronizes with the adversary by indicating to the adversary when the last round of AES decryption is performed. This allows the adversary to prime the cache immediately before the last decryption round is executed and probe it directly after it has finished.

Compared to these parallel works, the main benefit of our attack is that it requires no interrupts or synchrony assumptions, which makes it harder to detect and easier to deploy in practice.

Hähnel et al. [184] present a side-channel attacks against SGX enclaves aiming to improve temporal resolution of the side channel by frequently interrupting the enclave. They develop techniques to interrupt enclaves during the execution of instructions with memory operations, i. e., allowing them in observe each memory access of an enclave through cache monitoring. The increased temporal resolution allows side-channel attacks on a single execution of the victim enclave. Weiser et al. [405] show that the adversary's strong capabilities in the context of SGX, i. e., the adversary's capabilities to perform precise, low-noise side-channel attacks, enable new attack scenarios where just a single execution of the victim enclave is monitored. This endangers algorithms that are typically executed only once, for instance key generation algorithms, and were therefore considered unaffected by side-channel attacks in the past. However, such attacks are detectable by defenses such as T-SGX [353] or Déjá Vu [96] (cf. Section 4.4.2), while our attack was designed to circumvent these defense mechanisms. Apart from that, they use techniques similar to our attack (cf. Section 4.1) to minimize noise in the cache side channel.

SGX-Step is a framework that allows single-step execution of SGX enclaves [388], i. e., it allows the OS to schedule interrupts such that an enclave can execute one instruction before being interrupted and control is transferred back to the OS. This single-step execution of enclaves enables maximal temporal resolution for side-channel attacks, including cache-base and paging-based side-channel attacks.

DR.SGX is designed to thwart such side-channel attacks against SGX enclaves regardless of the attack technique used. We discuss its effectiveness and security properties in Section 4.3.5.

Speculative Execution Attacks. The Foreshadow attack allows, to extract the memory content of SGX enclaves using the processor's speculative execution unit to bypass the enclave memory's access control restrictions [390]. In combination with a side channel, e. g., cache side channels, the speculative execution unit can leak the enclave memory content to an adversary. SgxPectre also uses speculative execution to extract information from enclaves, however, it relies on the existence of vulnerable code gadgets in the enclave's binary [95]. These attacks can circumvent all knows side-channel defenses for SGX, however, Intel released microcode updates to mitigate these attacks [209], thus, up-to-date processors are not vulnerable to Foreshadow or SgxPectre.

4.4.2 *Side-Channel Countermeasures*

In this section, we discuss countermeasures against side-channel attacks proposed in related work. We elaborate on their applicability to protect SGX enclaves, discuss their effectiveness against our attack and compare them against our defense DR.SGX. Most of the discussed approached (except for Oblivious RAM (ORAM)) do not aim to protect against information leakage due to the usage of *external* resources, i. e., they are orthogonal to HardIDX.

Cache Disabling. The most straightforward countermeasure against cache-based side channels is to disable caching entirely [6]. This approach, however, defeats the main purpose of a cache, i. e., performance optimizations, resulting in severe performance degradation. Even more fine-grained approaches, for instance disabling caches only when security critical code is scheduled for execution, can be prohibitively expensive for many use cases. In the context of SGX, it would mean to disable caching during enclave execution. However, SGX enclaves may need to process large data-sets (e. g., human DNA) or perform expensive computation (e. g., cryptographic computations), or run large applications, e. g., the Haven library OS allows to load an entire Database Management System (DBMS) into an enclave [41]. Furthermore, disabling the cache cannot be enforced by an SGX enclave as the configuration of the system's caching behavior is controlled by the untrusted OS.

Architectural Changes to Cache Organization. Another approach to mitigate cache-based side channels is to introduce countermeasures through a redesign of the cache hardware. Respective techniques largely fall into two categories, the first one relies on access randomization within cache memory [399, 400, 218, 251], and the second one uses cache partitioning, so that security sensitive code never shares caches with untrusted processes [296, 297, 398, 399, 134].

Hardware approaches can also co-exist with software defenses. For instance, the Sanctum [110] architecture, which provides protected enclave execution for RISC-V platforms, applies cache partitioning for the LLC, while flushing the per-core L1 cache upon enclave exit.[11]

However, hardware changes can only be incorporated by hardware manufacturers, which is hard to achieve in practice. In particular, Intel SGX does not incorporate protections against side-channel attacks at the architectural level.

DR.SGX works on current processors and protects not only from cache-based side channels. It also protects against other side channels (e. g., based on page faults).

Obfuscation Techniques. ORAM [168, 169, 364, 379, 325, 170, 408] refers to schemes that hide the memory access patterns of a trusted client to an untrusted (encrypted) memory. It masks memory access patterns of the client by continuously shuffling and re-encrypting data as they are accessed in memory. ORAM assumes a server-client models where the trusted client accesses data from the untrusted server, typically via network. However, the scheme can also be applied to other scenarios, e. g., a trusted

11 Flushing is sufficient to ensure that L1 is never shared between an enclave and any other code on systems such as Sanctum that do not support Simultaneous Multithreading (SMT) (or hyper-threading).

CPU accessing untrusted memory such as Dynamic Random Access Memory (DRAM) or disk storage. ORAM requires the client to store state information that is updated throughout the execution. Oblivious execution architectures [258, 249, 250] attempt to hide all observable effects of program execution, including memory accesses (to code and data) and timing information. Implementing ORAM for every enclave memory access is extremely expensive. Obfuscuro [11] implements both ORAM and oblivious execution, with performance overhead of 83 × on average and up to 220 × in some cases. DR.SGX's performance overhead is at least one order of magnitude lower than Obfuscuro.

Sinha et al. [356] propose a compiler-based tool to protect code written in their custom language from paging-based side-channel attacks. In contrast, DR.SGX works with existing code in C/C++. Additionally, it mitigates cache-based side-channel attacks.

Raccoon [323] is a system that provides oblivious data access only for developer-annotated enclave data, thus reducing overhead. Memory accesses are hidden by either using ORAM or by streaming over the entire data structure. In contrast, DR.SGX does not rely on developers to identify and annotate potentially sensitive data.

ZeroTrace [337] is an oblivious data structure framework for SGX that runs on top of a software memory controller. ZeroTrace is designed to hide memory access to resources *outside* of an enclave, e. g., to the hard disk drive. Importantly, it is not designed to make *all* memory accesses of an enclave to main memory oblivious, unlike DR.SGX. Thus, it does not protect against cache-based side-channel attacks, for instance our attack presented in Section 4.1. Furthermore, ZeroTrace requires the developer to use the memory controller interface for all access that should be protected. DR.SGX does not require similar developer assistance.

Using ORAM to hide access patterns of searchable encryption schemes, is not straightforward, as has been shown by Naveed [281], and requires special ORAM techniques, such as TWORAM [154].

Shinde et al. [354] propose a compiler-based approach to hide memory page access pattern, i. e., making enclaves page-fault oblivious, by ensuring that the same memory accesses occur for all inputs. Their approach causes an overhead of 700 × on average (and 7000 × in the worst case) without developer's assistance while supporting only a subset of C/C++. DR.SGX, in contrast, is agnostic to the programming constructs used in an enclave to be protected, provides better performance without developer assistance, and provides protecting against all side-channel attacks aiming to leak information from memory accesses.

Other obfuscation techniques perform periodic scrubbing and flushes of shared caches [418] or add noise to memory accesses [296, 294] to interfere with the signal observable by the adversary. These techniques, however, introduce a significant overhead and are less effective on systems supporting SMT, where two threads or processes can be executed simultaneously, not in a time-sharing fashion. In this case, the attacker process running in parallel with the victim can still observe memory access patterns between scrubbing and flushing rounds. Furthermore, an adversary may collect multiple execution traces and process them to filter out the injected noise.

In response to the Foreshadow [390] attack, Intel provided microcode updates for SGX-enabled CPUs that cause a L1 cache flush on each enclave exit [209]. To be effective, SMT must be disabled on the target system, resulting in performance degradation.

Application Hardening. Application-level hardening techniques modify application code to protect secrets from side-channel leakage. As a countermeasure, Brickell et al. [75] proposed a technique called *scatter-gather*, that interleaves the multipliers of a cryptographic algorithm in memory and ensures that always the same cache lines of the pre-computed table are accessed irrespective of the accessed multiplier. Such solutions can be classified into two categories: (i) Side-channel free implementations (e. g., for cryptographic algorithms, such as AES and RSA [75, 231]) and (ii) automated transformation tools that can be applied to existing programs [106, 102, 113]. Side-channel free implementations, such as scatter-gather, are application-specific and require significant manual effort and expert knowledge about side-channel attacks (all application developers cannot be expected to be security experts). While side-channel resilient frameworks exist, e. g., data-oblivious machine learning algorithms [291], genome sequencing algorithms [263] and a side-channel resilient MapReduce framework [290] for SGX, they do not represent a general solution. These defenses are tailored to specific enclaves and algorithms. On the other hand, approaches that rely on automated compiler transformations are either probabilistic [106], i. e., making attacks harder but not impossible, or target only a specific type of side-channel attacks, such as execution-time-based attack [102, 113].

Randomization. Address Space Layout Randomization (ASLR) [305] is a common defensive technique against memory corruption attacks such as Return-Oriented Programming (ROP) [328]. ASLR hides the locations of memory regions (code and data) by randomizing their offsets at load time. More fine-grained solutions randomize exclusively code (not data) at function [220], block [402, 122], or instruction [298, 191] level.

Seo et al. [341] proposed the SGX-Shield framework that enables code randomization for SGX enclaves. While the primary goal of SGX-Shield is to protect enclaves from exploitable software bugs, the authors mention that randomization imposes additional burden to side-channel attacks, and in particular, it provides reasonable protection against page-fault side-channel attacks, as it forces an adversary to brute force 2^7 times in order to identify a single address value. However, this argumentation does not directly apply to our attack, because SGX-Shield concentrates on randomization of code, it does not randomize data. Hence, SGX-Shield cannot hide data access patterns leveraged in our attack. More generally, randomization of data segments is challenging due to dynamic data allocations, large data objects (e. g., tables) that need to be split up and randomized, and pointer arithmetic, which is typically used to access parts of large data objects (e. g., base-pointer relative offsets are often used to access table entries).

These challenges, among others, have been overcome by DR.SGX.

Software Diversity. Crane et al. [113] apply dynamic software diversity, an effective countermeasure against code reuse attacks and reverse engineering, to defend against cache-based side-channel attacks. Their approach is to create multiple copies of a program's code and choose one of them at the time of execution. We apply this technique to protect stack data in DR.SGX. However, the solution by Crane et al. is specifically

targeting the protection of cryptographic algorithms. In contrast, DR.SGX can protect non-cryptographic algorithms, as well.

Attack Detection. Previous works [306, 100] suggested to use system-level monitoring of HPCs to detect cache performance anomalies as a signature of ongoing cache-based attacks. However, this method is not applicable in the context of SGX's adversary model, since an adversary has sufficient privileges to disable any monitoring at system level.

T-SGX [353] and Déjá Vu [96], propose detection methods for side-channel attacks that are based on frequent interruption of the victim enclave. A prime example of such *privileged* attacks is the deterministic side channel based on page-faults [412]. Here, the OS incurs page faults during enclave execution and learns the execution flow or data access patterns of the enclave from the requested pages. T-SGX and Déjá Vu suggest using a hardware implementation of *transactional memory* in Intel processors called Transactional Synchronization Extensions (TSX), to notify an enclave about exceptions, e. g., page faults, without interference by the system software.

T-SGX [353] modifies the enclave code such, that any interruption is detected and execution is terminated. However, this approach requires, in order to be effective, that the enclave cannot be restarted after the attack attempt was detected. To achieve this, T-SGX requires one-time tokens provided by an external party over a secure channel. If the adversary targets the cryptographic protocol used in the establishment of that secure channel, this condition cannot be enforced (the adversary can initiate and replay the protocol). Once the adversary has extracted a valid token, the adversary can misuse it to run the victim enclave arbitrarily often, i. e., extract information despite the self-termination of the enclave.

Déjá Vu [96] extends enclave programs with execution time checks in order to detect delays caused by interruption of the enclave. SGX does not provide a reliable, fine-grained time source to enclaves, therefore, Déjá Vu uses a "counting thread" as a timer. The timer thread guards itself from being interrupted through the use of TSX, similar to T-SGX. While cache eviction might slow down the victim, leading to a detectable delay, the timer thread can be slowed down as well, without interrupting it. The authors acknowledge that the timer thread can run on a CPU core clocked at the lowest frequency setting while the victim runs on a CPU core set to maximum frequency. This can lead to a discrepancy of factor two or more, while, based on our experiments, L1 cache eviction due to constant priming slows down the victim only by 27 %.

These defenses do not prevent our attack presented in Section 4.1 and other attacks that work without interrupting the victim enclave [340, 174]. DR.SGX, in contrast, is applicable to all side-channel attacks that leak memory access patterns.

Cloak [182] uses TSX to perform atomic memory operations that hide sensitive memory accesses. Before sensitive memory is accessed, all cache lines are touched (primed) by the enclave, and thus the adversary learns nothing about the enclave's sensitive accesses because the adversary cannot distinguish cache lines that were primed from those that were actually used by the enclave. Cloak relies on the developer to annotate sensitive data structures that should be protected from side-channel attacks and requires TSX, which is not supported in all SGX processors. DR.SGX does not require similar developer assistance and works on all SGX processors. Völp et al. [394] use a concept called *delayed*

preemption [386], which can be implemented in hardware or in software using a trusted hypervisor, to allow an enclave to execute uninterrupted or to be informed in case an interrupt occurred during execution. By priming (i. e., pre-loading) the TLB, enclaves can execute without requiring the OS to load memory on-demand during execution. Therefore, a benign OS does not need to interrupt an enclave's execution, and thus, any interrupt occurring during the enclave's execution indicates an attack. This defense is not effective against our attack, which does not cause interrupts of the enclave. DR.SGX provides a software-only defense that can be used on deployed systems without changing the hardware or requiring assistance from privileged software layers. Furthermore, DR.SGX provides protection against all side-channel attacks extracting memory access patterns, also those that do not interrupt enclave execution.

Cooperative Privileged Software. HyperRace [94] and Verys [292] aim to prevent same-core side-channel attack, i. e., side-channel attacks that rely on SMT to run the victim enclave and the attacker code in parallel on the same physical CPU core and exploit core-exclusive resources such as the L1 cache. The idea of these solution is that an enclave occupies both SMT execution units of a CPU core by running two threads in parallel. Since the untrusted OS is responsible to schedule both enclave threads on the same CPU core, the enclave threads have to verify that they are indeed co-located. HyperRace uses contrived data races on a shared variable between the two enclave threats while Verys measures cache access timings to check co-location. These defenses would be effective against our attack, which utilizes SMT to monitor an enclave's memory access patterns. However, HyperRace and Verys, thwart only side-channel attacks that rely on CPU core co-location while DR.SGX provides a general approach to counter all types of side-channel attacks aiming to leak memory access patterns.

Database Protection Approaches. A number of approaches have been developed that specifically aim at protecting (outsourced) databases from leaking information. They use different techniques and approaches, such as searchable and property-preserving encryption or TEEs, to achieve this goal.

Searchable Encryption. Range queries are supported only by a small subset of all searchable encryption schemes proposed in related work [58, 396, 352, 257, 128, 144]. All existing schemes supporting range searches have in common that they leak the access pattern, including our solution HardIDX. However, all other schemes, except the one by Shen et al. [352], leak the search pattern or the order of stored database entries. HardIDX does not leak either of them, and is significantly more efficient, i. e., HardIDX is the first scheme with polylogarithmic search time that leaks only the access pattern. A detailed comparison of the different schemes' complexities is provided in our conference publication [149] and journal article [151]. Other schemes focus on other problems, such as multi-dimensional range queries [385, 410] or query executions by multiple parties without the need for these parties to share an encryption key [99].

Encrypted Databases. Encrypted databases [314] rely on property-preserving encryption [10, 56, 57, 219], which enables the database engine to use the same internal index structures as for plain-text data, to achieve low run-time overhead. While this approach enables efficient operations, including range queries, Naveed et al. [282], and follow-up works [136, 179],

showed practical attacks against property-preserving encryption schemes, which can recover plain-text data with high probability in many cases.

Other approaches enabling range queries include an extension of the OXT protocol [87] by Faber et al. [144], as well as different approaches presented by Demertzis et al. [128].

TEE-protected Databases. Maheshwari et al. [260] proposed a Trusted Database System (TDB) that uses a TEE to isolate an entire database when operated in a hostile environment, leading to a very large Trusted Computing Base (TCB) of this solution. TDB encrypts all data and metadata of the database when stored in untrusted memory, however, it is not concerned with information leakages, e. g., due to access patterns to external resources or side channels.

CryptSQLite [397], STANlite [336] and EnclaveDB [318] protect databases using SGX, STANlite aims to increase the performance of SGX-protected databases [336], in particular, in the face of limited enclave memory. Also, ShieldStore [222] aims to overcome the restrictions SGX's limited enclave memory poses in the context of key-value stores. These approaches, unlike HardIDX, are not concerned with access pattern leakage or side channels. StealthDB is a DBMS using SGX enclaves to execute queries in isolation [177]. StealthDB leaks information about the order of data elements that are accessed when serving search queries.

Cheng et al. [99] propose an SGX-based scheme for performing search over encrypted data, which does not leak access patterns. However, their scheme's complexity is linear in the size of the search tree while HardIDX achieves logarithmic complexity.

Practical Oblivious Search and Update Platform (POSUP) leverages ORAM to prevent information leakage [192], however, it only supports keyword searches. Mishra et al. [276] propose Oblix, a protected search index using enclaves [276]. Oblix uses ORAM to hide accesses to data stored outside of enclaves, while protecting the internal processing of data against leakage at memory-page granularity. Oblix goes beyond HardIDX as it supports updates of the search index, i. e., insertion and deletion of data, and hides the size of the search result.

5 TRUSTED EXECUTION ENVIRONMENT APPLICATIONS AND USE CASES SCENARIOS

Trusted Execution Environments (TEEs) enable the protected processing of overly sensitive data, such as personal data or corporate secrets. The cloud is an natural usage scenario for TEEs as secret data needs to be protected when outsourced to untrusted infrastructure. When data-processing is outsourced control over the data is given to the cloud provider, which cannot always be fully trusted. On one hand, the attack surface in the cloud is enlarged, e. g., due to the risk of insider attacks of malicious cloud administrators or malicious co-tenants. On the other hand, legal requirements, such as data protection regulation [142], prohibit the unrestricted usage of cloud computing for many use cases.

TEEs allow protected computing in hostile environments, and thus, enable new applications in the context of cloud computing. In Section 5.1 we present VoiceGuard [71], a solution that demonstrates how Intel's Software Guard Extensions (SGX) can be used to protect sensitive user input data as well as Intellectual Property (IP), i. e., machine learning models, for speech recognition systems.

Scenarios and use cases where two mutually distrusting parties provide sensitive inputs for a joint computation are quite common. In our conference publication "Secure Multiparty Computation from SGX" [315] we demonstrate how to utilize SGX's isolated execution in combination with its Remote Attestation (RA) functionality in order to construct a general secure execution framework for multiple mutually distrusting input parties.

While enclave-like, i. e., unprivileged, TEEs enable many novel applications, they do not fulfill the requirements for all usage scenarios. For some security solutions, the TEE is required to have extended privileges to control untrusted software. In Section 5.2 we present a policy enforcement mechanism for smart mobile devices, for instance smartphones, allowing their use in restricted spaces, such as federal offices or companies, according to constrains defined by the host [64]. Our enforcement mechanism relies on ARM TrustZone's capabilities to inspect and modify a devices entire memory in order to assert device-usage policies.

5.1 VoiceGuard

Voice User Interfaces (UIs) are becoming common-place, e. g., through the integration of smart assistants, such as Amazon Alexa, Apple's Siri, Google assistant and Microsoft's Cortana, in various appliance, for instance smartphones, cars and smart-home devices. The user's voice input is used for speaker authentication and to control these systems via Automated Speech Recognition (ASR). For these services, the user's voice data is sent to the cloud for processing, posing a significant risk, as voice data is highly sensitive. It contains the spoken words and other ambient sounds that might reveal information not meant to be revealed to outsiders. Past incidents have shown that the service providers attain user data, e. g., to improve their machine learning models [187, 124]. Furthermore, voice data contain biometric information of the speaker, which allow the identification of the speaker and can be abused to impersonate the speaker, i. e., create fake audio recordings of a person with arbitrary content.

Hence, the user's voice input data must be protected. The naïve approach, i. e., performing speech recognition on the user's device, is not practical since the algorithms and models used to process the data are high value Intellectual Property (IP) of the service providers. Therefore, they are unwilling to impart their IP to users' devices without protection guarantees.[1]

This situation calls for a solution that protects both the user's inputs as well as the IP of the service provider. Cryptographic approaches, such as Secure Multiparty Computation (SMC) or Fully Homomorphic Encryption (FHE), induce high computation and communication cost rendering them impractical. TEEs enable the protected processing of the user's input data without revealing it to the cloud provider, and, at the same time, they enable the protection of the service provider's IP.

Goals and Contributions. Our goal is an efficient speech recognition framework that used TEEs to provide real-time processing of voice data. It must protect both, user voice inputs as well as the algorithms and models used to process these data. Furthermore, the solution should support user-specific models that adapt to each user's specific characteristics such as varying pronunciations, accents, and dialects.

Our privacy-preserving speech recognition framework, called VoiceGuard, makes the following main contributions:

- We present the design of VoiceGuard, our TEE-based ASR framework protecting both, the user's voice input data as well as the speech recognition model. VoiceGuard's design can be easily extended to support user-specific models, using techniques such as feature transformation, i-vectors or model transformations.
- We implement VoiceGuard using Intel's Software Guard Extensions (SGX) and the Kaldi ASR toolkit [316].

1 In this work we focus on cloud-based solutions. Given the availability of Trusted Execution Environments (TEEs) similar schemes apply for other systems as well; Bayerl et al. [42] present a speech recognition solution for user devices based on Sanctuary (see Section 5.3).

- We evaluate VoiceGuard's performance based on the *Resource Management* and *WSJ* speech recognition tasks, demonstrating its capability to perform real-time speech recognition.

5.1.1 *Model and Assumptions*

In this section we consider a setting where three parties collaborate to perform secure and private speech processing:

(1) The *user* $\mathcal{U}$ provides the voice data to be processed. $\mathcal{U}$ is concerned about the privacy of the provided data. The other parties should not be able to identify $\mathcal{U}$ based on biometric characteristics in the input data. Additionally, $\mathcal{U}$ does not want to reveal the content of the input data to the other parties, i. e., they should not be able to access the voice data or the processing results. Lastly, $\mathcal{U}$ does not want to be traceable across multiple sessions.

(2) The *vendor* $\mathcal{V}$ provides the software required for speech processing together with corresponding models. This data constitutes $\mathcal{V}$'s IP; hence, it must be kept confidential from the other parties.

(3) The *service provider* $\mathcal{SP}$ carries out the actual computations based on $\mathcal{U}$'s and $\mathcal{V}$'s inputs. $\mathcal{SP}$ could be an independent third party, e. g., a cloud service provider. Without loss of generality, $\mathcal{SP}$ could also be under the control of $\mathcal{U}$ or $\mathcal{V}$.

Adversary Model. The adversary $\mathcal{ADV}$'s goal is to extract sensitive information, i. e., the intellectual property of $\mathcal{V}$, the input of $\mathcal{U}$, or data that allows $\mathcal{ADV}$ to identify or track $\mathcal{U}$.

We assume that $\mathcal{ADV}$ is in control of $\mathcal{SP}$'s infrastructure, in particular, all computer systems involved in performing the speech processing task. $\mathcal{ADV}$ has full control over the software in $\mathcal{SP}$'s infrastructure, including privileged software such as the Operating System (OS) or a hypervisor.

We assume that $\mathcal{SP}$'s infrastructure allows isolated processing of data in a TEE. We assume that $\mathcal{ADV}$ cannot directly access the code and data processed inside a TEE. $\mathcal{ADV}$ cannot perform invasive hardware attacks such as extracting keys from the Central Processing Unit (CPU). We also consider physical side-channel attacks, for instance differential power analysis [224], out of scope. Further, we assume that the developer of the speech processing software incorporates appropriate defense mechanisms in the software executing insider the TEE in order to protect it against side-channel attacks leveraging micro-architectural effects [353, 96, 66].[2]

5.1.2 *VoiceGuard Design*

Our architecture VoiceGuard enables privacy-preserving and efficient speech processing on untrusted systems. VoiceGuard supports different deployment scenarios, i. e., the service provider $\mathcal{SP}$ is not necessarily a third party, e. g., $\mathcal{SP}$ could also be the user $\mathcal{U}$ or the vendor $\mathcal{V}$. Common to all scenarios is the basic setup, i. e., at least two *input parties*

2 Our evaluation is performed without such protection mechanisms, such as DR.SGX presented in Section 4.3, and thus does *not* reflect their impact on the performance results.

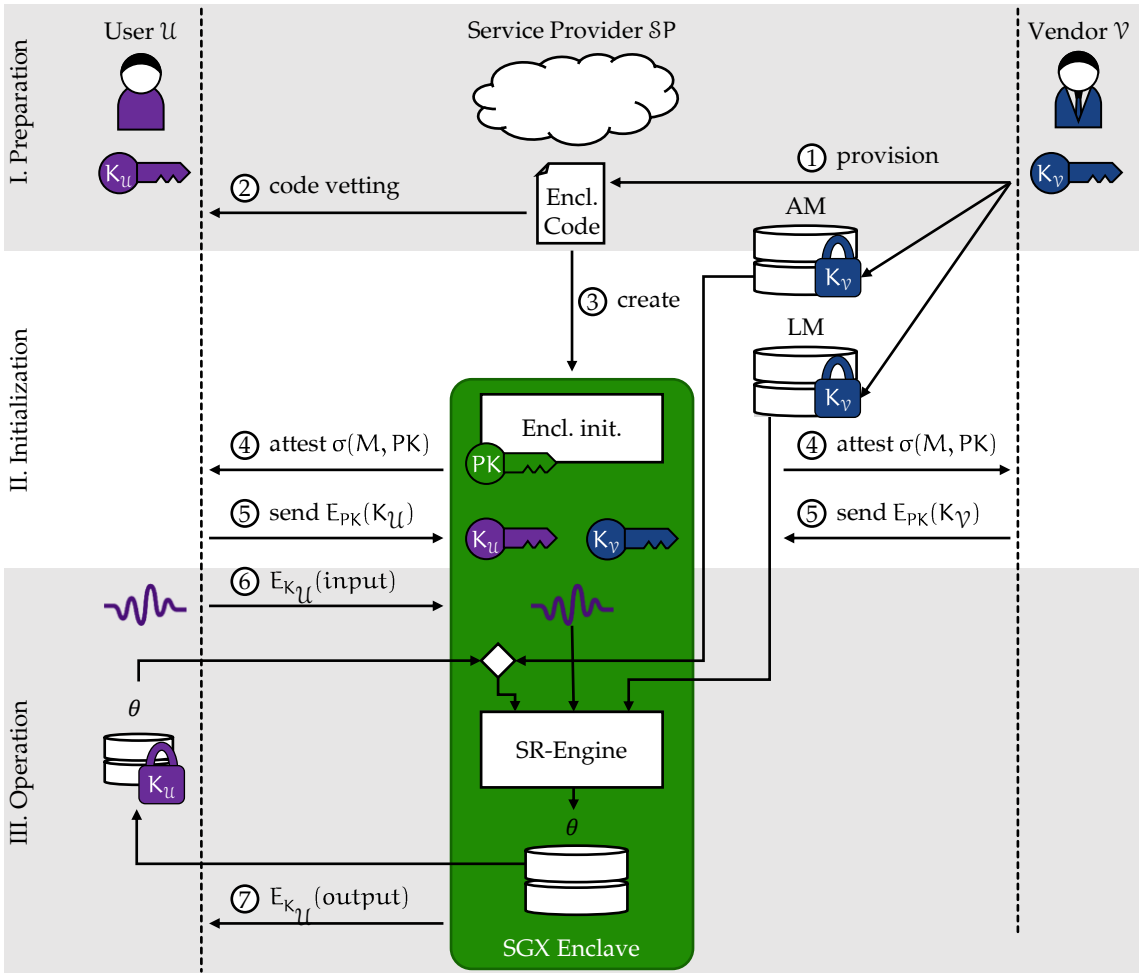


Figure 31: VoiceGuard architecture. User $\mathcal{U}$ establishes a secure channel with the SGX enclave hosted at service provider $\mathcal{SP}$ and sends sensitive voice data as well as user-specific adaptation data θ. Similarly, vendor $\mathcal{V}$ sends the sensitive models *AM* and *LM* through a secure channel. $\mathcal{SP}$ securely processes $\mathcal{U}$'s voice data using $\mathcal{V}$'s models within an SGX enclave.

provide sensitive data while the computing platform is not trusted by at least one of the input parties.

For the sake of simplicity, we explain our solution based on the speech recognition scenario visualized in Figure 31, where the service provider $\mathcal{SP}$ is an untrusted third party, e. g., a cloud service provider. $\mathcal{SP}$'s infrastructure provides TEEs, which guarantee isolated execution. In this work we use Intel's SGX as a TEE instance. The vendor $\mathcal{V}$'s private input consists of speech recognition models. The user $\mathcal{U}$'s private input is the voice data. In this example, the output is sensitive as well and should only be made available to $\mathcal{U}$.[3]

VoiceGuard works in three phases: (I) preparation, (II) initialization, and (III) operation. In the first phase, user $\mathcal{U}$ and vendor $\mathcal{V}$ need to agree on the code to be executed in a TEE,

3 The output could also be provided to one or multiple other parties.

called enclave in the context of SGX, ("Encl. Code" in Figure 31). In the second phase, the enclave code is instantiated. $\mathcal{U}$ and $\mathcal{V}$ use Remote Attestation (RA) to establish secure channels with the enclave through which they provision their respective encryption keys to the enclave. In the third phase, the enclave is ready to perform speech processing. Using the keys transmitted in the previous phase, $\mathcal{U}$ and $\mathcal{V}$ provide their encrypted inputs to the enclave. The result of the operation phase is encrypted with $\mathcal{U}$'s key. Hence, only $\mathcal{U}$ can decrypt it and learn the output. Next, we describe the individual phases in detail:

Preparation Phase. First, $\mathcal{U}$ and $\mathcal{V}$ need to agree on the code to be run inside the SGX enclave. While SGX protects enclaves against accesses from the outside, enclaves are nevertheless allowed to output data without any restriction. Therefore, $\mathcal{U}$ and $\mathcal{V}$ want to make sure that the enclave code only outputs non-sensitive data. The code typically comes from $\mathcal{V}$, i. e., $\mathcal{V}$ provisions the enclave code, ① in Figure 31. Thus, $\mathcal{V}$ can easily ensure that no sensitive data will leave the enclave. The code itself is not necessarily confidential and is often open source. However, $\mathcal{U}$ has to carefully analyze the enclave code in a *vetting* process ② to verify that it does not contain functions that will leak $\mathcal{U}$'s sensitive data. The vetting process could also be outsourced to a trusted third party, e. g., a government institution.

Additionally, $\mathcal{V}$ provisions its acoustic model *AM* and language model *LM* to $\mathcal{SP}$. Both are encrypted with the $\mathcal{V}$'s key $K_{\mathcal{V}}$, hence, $\mathcal{SP}$ cannot access $\mathcal{V}$'s intellectual property. At this stage, the models are not yet loaded inside an enclave. The models are written to untrusted storage, e. g., the hard disk.

Initialization Phase. The SGX enclave is created from the code provisioned by $\mathcal{V}$ earlier, step ③. The creation process is measured by the SGX-enabled CPU, i. e., a cryptographic hash of the initial memory content of the enclave is created and stored securely. If the enclave code is manipulated before or during the creation process, the measurement will produce a different result and the manipulation will be detected through RA. After the creation is finished, the code is isolated from all accesses and cannot be changed anymore.

The first operation performed by the enclave is the enclave initialization, during which the enclave generates a key pair for asymmetric cryptography operations such as Rivest–Shamir–Adleman (RSA) [327],[4] with the public key *PK* shown in white in Figure 31.

Next, $\mathcal{U}$ and $\mathcal{V}$ need to establish a secure channel with the enclave by provisioning their keys $K_{\mathcal{U}}$ and $K_{\mathcal{V}}$, respectively, to the enclave. We will describe this process for $\mathcal{U}$. The process for $\mathcal{V}$ is identical. VoiceGuard uses a public key cryptography protocol similar to Transport Layer Security (TLS) [131], which is widely used to secure web sites. The enclave sends its public key *PK* to $\mathcal{U}$. However, $\mathcal{U}$ needs assurance that the received *PK* comes indeed from the correct enclave, i. e., the authenticity of *PK* must be established. This is done using the RA feature of SGX, which generates a digital signature $\sigma(M, PK)$ that binds *PK* to the measurement M of the enclave, ④ in Figure 31. In particular, the public key *PK*, which was generated *inside* the enclave, and the measurement of the initial

4 This process leverages the hardware Random Number Generator (RNG) of the CPU and can therefore not be influenced from outside the enclave.

enclave memory content are signed with the platform attestation key. This signature can be verified using Intel's Public Key Infrastructure (PKI) for SGX.

$\mathcal{U}$ verifies the signature and checks that M matches the measurement of the enclave agreed on with $\mathcal{V}$, i. e., that the enclave has not been altered before or during creation. If both checks were successful, $\mathcal{U}$ can be sure that PK belongs to the key pair generated by the correct enclave and that information encrypted with PK can only be decrypted inside that enclave. In step ⑤, $\mathcal{U}$ encrypts $K_{\mathcal{U}}$ with PK and sends the result $E_{PK}(K_{\mathcal{U}})$ to the enclave.

At the end of the initialization, the enclave shares a symmetric key with the user $\mathcal{U}$ ($K_{\mathcal{U}}$, the violet key in Figure 31) and with the vendor $\mathcal{V}$ ($K_{\mathcal{V}}$, the blue key in Figure 31).

Operation Phase. $\mathcal{U}$ sends encrypted inputs $E_{K_{\mathcal{U}}}(\mathtt{input})$, i. e., audio samples, to $\mathcal{SP}$. The input is encrypted with $\mathcal{U}$'s key, hence, it can only be accessed by the enclave ⑥. If applicable, $\mathcal{U}$ also sends user-specific adaptation parameters θ (e. g., i-vectors), which are also encrypted with $K_{\mathcal{U}}$, to the enclave.

Inside the enclave, $\mathcal{U}$'s input is decrypted and passed to the speech recognition engine ("SR-Engine" in Figure 31). The SR-Engine has two additional inputs, the Acoustic Model (AM), typically a Deep Neural Network (DNN), and the Language Model (LM), typically a decoding graph. AM is provided by $\mathcal{V}$ and already stored encrypted at $\mathcal{SP}$. When AM is used, it is loaded into the enclave and decrypted using $\mathcal{V}$'s key $K_{\mathcal{V}}$. Similarly, any adaptation parameters θ and the LM are loaded by the enclave, decrypted, and passed to the SR-Engine.

On-demand loading of AM or LM could leak sensitive information about their structure by observing access patterns. This can be prevented by storing this data in a randomized order, i. e., preventing an observer from learning useful information from observed access patterns, similar to the techniques presented in Section 4.2.

The result of the speech processing is encrypted with $K_{\mathcal{U}}$ and sent back to $\mathcal{U}$ ⑦.[5] Additionally, the SR-Engine may produce updated adaptation parameters θ, which are then encrypted with $K_{\mathcal{U}}$ and sent back to $\mathcal{U}$.[6]

Once in the operation phase, the system can be queried repetitively by $\mathcal{U}$, thereby avoiding repeated preparation and initialization costs.

5.1.3 *Implementation and Evaluation*

We implemented and evaluated VoiceGuard on an Intel Core i7-7700 CPU (3.60 GHz) with 128 MB Enclave Page Cache (EPC). We encapsulated Kaldi [316] in an SGX enclave, supported by the Graphene library OS [384]. We evaluated VoiceGuard's run-time overhead compared to unprotected, i. e., outside of an enclave, executed Kaldi instance on the same platform, using two representative corpora: DARPA Resource Management (RM) [317] and Wall Street Journal (WSJ) [155].

Our experiments show that VoiceGuard's overhead for recognizing text from audio data ranges between 39 % and 49 % for the RM data set. For the larger WSJ data set

5 The result could also be sent to a different party, even a third party.
6 $\mathcal{U}$ can decrypt θ and re-encrypt it to make individual requests from the same user unlinkable.

VoiceGuard introduces overhead between 98 % and 104 %. Although the processing time – in the worst case – doubles, the performance of VoiceGuard nevertheless enables privacy-preserving speech processing in real time.

More evaluation details for VoiceGuard are provided in our conference publication [71].

5.1.4 *Conclusion*

VoiceGuard is a novel architecture for efficient privacy-preserving speech recognition. It can be deployed either on-premises, in the cloud or on the user devices, and supports user-specific models. We demonstrated the feasibility of using SGX enclaves for machine learning tasks. We showed that practical performance can be achieved when considering SGX's memory limitations when selecting and configuring existing speech recognition frameworks. VoiceGuard is applicable for speech recognition in real time, as shown in the evaluation of our prototype implementation. VoiceGuard's generic architecture allows its use for other tasks than speech recognition, for instance speaker verification or voice biometrics, including emotion recognition and medical speech processing.

5.2 Regulating ARM TrustZone Devices in Restricted Spaces

Smart personal devices, such as smartphones, come with a wide array of peripheral sensors and interfaces that enable their extensive misuse in various environments. They can be misused to exfiltrate sensitive information, e. g., from government institutions or enterprises. Similarly, smart devices can be used to gain unfair advantages, e. g., by smuggling unauthorized information into examination halls.

Such misuse is typically prohibited, either explicitly, e. g., by laws and regulations, or implicitly, e. g., by social norms. However, the enforcement of these usage policies is an open problem. Currently the threats of misuse are countered by ad-hoc methods. For instance, personal smart device must not be brought into some federal offices, which is enforced by physical checks before entering. In companies personal devices often are not permitted to connect to the company network and employees are forbidden to store corporate data on their personal devices. And in examination settings smart devices typically have to be turned off and stored away.

These traditional solutions are particularly unpractical when smart functionalities get integrated into everyday items, e. g., clothes or medical devices. For instance, asking an individual to refrain the usage of prescription glasses with smart features (or other assistive health devices) is impractical. Therefore, a solution is needed that allows the targeted regulation of smart devices' functionalities.

Goals and Contributions. The goal of this work is to develop methods to regulate the usage of smart devices in restricted spaces. Our solution should selectively allow/prohibit the use of smart device functionalities based on a policy defined by a restricted space's host. It must enable the host to retrieve a proof that its usage policies is enforced on the device, and that it was enabled for as long as the device was within the restricted space. The size and complexity of the policy-enforcement mechanism on the guest device must be minimized and protected by strong isolation mechanisms. At the same time, the guest must be able to validate the policies enforced by the host. Furthermore, the gust must be able to disarm and revert the host's policies after leaving the restricted space.

This work makes the following main contributions:

- We provide a novel method, which allows hosts to verify that a guest device complies with its policy. Leveraging ARM TrustZone a guest device provides unforgeable cryptographic *verification tokens* ensuring to the host the correct state of the guest device.
- We use host-initiate *remote memory operations* as a generic method to regulate guest device, i. e., to enforce policies on guest devices.
- We consider potential malicious hosts and developed a *vetting service* allowing guests to check that a host's policy does not violate the guest's security or privacy.
- We present a prototype implementation and evaluation of our solution showing that our solution allows fine-grained policy enforcement relying only on a small

policy-enforcement code base executing in the devices TrustZone Trusted Execution Environment (TEE).

5.2.1 *System Model, Assumptions and Requirements*

In this section we introduce the term *restricted spaces* followed by our adversary model.

5.2.1.1 *Restricted Spaces*

The increasing capabilities of mobile devices, in particular smartphones, and their ubiquitous connectivity pose an increasing risk of misuse in various environments, which we refer to as *restricted spaces*. In enterprises or government institutions, smart devices can be misused to ex-filtrate sensitive data, either by a malicious device owner or unwittingly by the device's owner, e. g., when the device is infected with malware. Also, smart device can be misused to infiltrate unauthorized information [21] or to collude with accomplices [273], e. g., to cheat in tests or exams. Furthermore, social conventions might prohibit the use of certain functionalities of smart devices, e. g., recording of video or audio during private conversations. All locations where such retrenchments are relevant represent restricted spaces.

5.2.1.2 *Adversary Model*

The adversary $\mathcal{ADV}$'s goal is to operate a device D in a restricted space without complying to the policies of the host $\mathcal{H}$. $\mathcal{ADV}$ is in control of all software on a guest $\mathcal{G}$'s device $D_{\mathcal{G}}$ except for software and data explicitly protected, i. e., the software and data in device's TEE. Hence, $\mathcal{H}$ does not trust software of a guest device $D_{\mathcal{G}}$. The software of $D_{\mathcal{G}}$ is, among other things, responsible for configuring and driving peripherals. However, $\mathcal{H}$ can establish trust in a $D_{\mathcal{G}}$'s (privileged) software after explicitly checking it, i. e., inspecting the state of $D_{\mathcal{G}}$.

The software and data protected in the TEE are trusted by all parties, i. e., the host $\mathcal{H}$ as well as the device owner $\mathcal{G}$. The TEE has exclusive access to cryptographic credential allowing it to uniquely authenticate itself. Similarly, $\mathcal{H}$ possesses cryptographic credential allowing it to authenticate. $\mathcal{H}$ can use a TEE to protect its cryptographic credentials as well. We assume that $\mathcal{H}$ is immune to attacks. Different entities, i. e., D and $\mathcal{H}$, can authenticate each other via a trusted third party, e. g., a Public Key Infrastructure (PKI)

The device owner $\mathcal{G}$ does not trust $\mathcal{H}$ and will not allow $\mathcal{H}$ to make arbitrary changes to $D_{\mathcal{G}}$.

Software attacks on $D_{\mathcal{G}}$ that cannot be detected by $\mathcal{H}$ via state inspection, e. g., publicly unknown attacks, such as zero-day exploits, are considered out of scope. Denial-of-Service (DoS) attacks are not prevented, however, they can be detected by $\mathcal{H}$, i. e., $\mathcal{ADV}$ can interrupt the communication of $D_{\mathcal{G}}$ with $\mathcal{H}$'s policy server at any point in time. $\mathcal{ADV}$ cannot use physical attacks to compromise the isolation guarantees of the TEE.

We consider covert uses of smart devices in restricted spaces an orthogonal problem. Hence, guests stealthily smuggling devices into the restricted space without check-in are assumed to be covered by traditional physical security methods.

5.2.1.3 *Requirements*

To enforce rules and policies for restricted spaces all devices $D_{\mathcal{G}}$ have to be checked-in when entering a restricted space. Further, a dedicated check-out process is needed to restore D's state when leaving the restricted space. $\mathcal{H}$, i. e., the entity in control of the policies relevant in a restricted space, has to operate a policy server that can deploy the policies to $D_{\mathcal{G}}$ and is responsible for validating the correct enforcement of these rules.

Furthermore, each guest device $D_{\mathcal{G}}$ has to be equipped with a TEE. All software running inside the TEE is trusted, i. e., it is part of D's Trusted Computing Base (TCB). This means the integrity and confidentiality of code and data inside the TEE must be preserved. All software that is not protected by the TEE – also dubbed outside of the TEE – is assumed to be untrusted, i. e., potentially compromised.

Check-in. When entering a restricted space $\mathcal{G}$ has to perform the check-in procedure for all its devices by connecting each $D_{\mathcal{G}}$ to $\mathcal{H}$'s policy server, which performs the following operations:

Authentication. First, $\mathcal{H}$, more precisely, $\mathcal{H}$'s policy server, and each guest device $D_{\mathcal{G}}$ have to mutually authenticate via a standard cryptographic scheme, e. g., using Transport Layer Security (TLS) [131].

$\mathcal{G}$'s authentication secret has to be protected on $D_{\mathcal{G}}$, i. e., it must be only accessible from the $D_{\mathcal{G}}$'s TEE. The policy server must be able establish a secure channel to $D_{\mathcal{G}}$'s TEE, protected from all potentially malicious software present on $D_{\mathcal{G}}$ outside of the TEE.

Furthermore, $\mathcal{H}$ must be able to ensure that the secure channel is established with the correct guest device to prevent relay attacks, such as the cuckoo attack [300].

Examination of Guest State. In the second step, $\mathcal{H}$ needs to obtain the current state of $D_{\mathcal{G}}$. The state is defined by the current memory and register content of software and data outside of the TEE.

However, the current state of $D_{\mathcal{G}}$ can be highly sensitive as the memory might contain personal data, cryptographic secrets, etc. Hence, $D_{\mathcal{G}}$ has to ensure that it protects itself from a malicious host, i. e., validate the legitimacy of the policy server's request.

$\mathcal{H}$'s policy server has to inspects the received state to validate that $D_{\mathcal{G}}$ is in a benign state and can be trusted as a whole. Furthermore, the policy server has to extracts relevant information about $D_{\mathcal{G}}$'s configuration, e. g., software versions and available peripherals. Additionally, $\mathcal{H}$ can store the receive memory snapshot for it to be restored at check-out.

If $\mathcal{H}$'s policy server is unable to validate the state of $D_{\mathcal{G}}$ or detects that $D_{\mathcal{G}}$ is compromised, the check-in has to be considered failed. In this case, $\mathcal{H}$ has to be notified, allowing it to resort to traditional measures, such as locking $D_{\mathcal{G}}$ away.

Policy Deployment. $\mathcal{H}$'s goal is to ensure that each $D_{\mathcal{G}}$ is in a state that conforms with its policies. In particular, $\mathcal{H}$ needs to enforce that particular peripherals of a smart device, for instance cameras, microphones or cellular network modem, are deactivated. The TEE on $D_{\mathcal{G}}$ has to enforce the restrictions on the peripheral devices.

Continuous Validation. After the modifications, which are necessary to comply with $\mathcal{H}$'s policies, have been applied the continues compliance of $D_{\mathcal{G}}$ must be verified, i. e., that the modifications are not reverted or circumvented. For this, the TEE service that applied

$\mathcal{H}$'s rules has to generate a verification token that is provided to $\mathcal{H}$. The token captures the state of $D_{\mathcal{G}}$ when $\mathcal{H}$'s policies are enforced, i. e., it provides Remote Attestation (RA) functionality.

Enabling $\mathcal{H}$ to request a new token at any point in time, allows the host to validate that $D_{\mathcal{G}}$ is still in the state that enforces $\mathcal{H}$'s policies. To prevent replay attacks the token needs to include a fresh nonce for each request from $\mathcal{H}$.

Check-out. After check-out, $D_{\mathcal{G}}$ should be able to function completely unrestricted. The check-out process needs to accomplish two objectives.

Guest State Validation. $\mathcal{H}$ needs to validate that $D_{\mathcal{G}}$ still is policy compliant. To ensure this $\mathcal{H}$ has to request a fresh verification token from $D_{\mathcal{G}}$.

Guest State Restoration. Finally, $D_{\mathcal{G}}$ should be restored to a state that lifts all restrictions. To achieve this $D_{\mathcal{G}}$ either has to be brought to a defined, unrestricted state, e. g., by rebooting the device, or by returning to the state initially captured during check-in. Alternatively, the changes that were applied during check-in have to be reverted explicitly, i. e., by applying the "inverse" operations of those that were performed during check-in to enforce policy-compliance.

5.2.2 *Design*

Subsequently we describe the design of our solution to restrict smart device D in restricted spaces. Our design is focused on D equipped with ARM's TrustZone security architecture, which is described in detail in Section 2.3.1. However, other security architectures fulfilling the necessary requirements can be used as well.

Our solution for regulating D in restricted spaces uses a primitive, which we denote *remote memory operations*. We will introduce this concept in Section 5.2.2.1.

Afterwards we introduce our design for regulating ARM TrustZone device in restricted spaces in Section 5.2.2.2. It uses ARM TrustZone's capabilities and the remote read/write primitive to enable the remote enforcement of a restricted space's policies and the validation of the enforcement. Our design is based on a validation and enforcement service running in the TrustZone secure world, which has the privileges to inspect and manipulate the memory of a device's normal-world memory, i. e., enables remote memory operations. According to our adversary model this service is trusted by $D_{\mathcal{G}}$'s owner $\mathcal{G}$ as well as the host $\mathcal{H}$ (cf. Section 5.2.1.2).

5.2.2.1 *Remote Memory Operations*

Our remote memory operation primitive allows a remote entity to perform read and write operations on a device D with the assurance that these operations were executed correctly, even if D's software is compromised. The primitive is based on a trusted proxy, which receives requests for read or write operations from the remote entity and performs them locally. Our trusted proxy service utilizes ARM's TrustZone to ensure the correctness of these operations, i. e., the integrity and authenticity of remote memory operations.

To prevent the misuse of remote memory operations all requests from remote entities are authenticated and integrity protected. Once a request is received by the proxy service, its origin is validated. If it comes from an untrusted entity, it is discarded. Similarly, if the integrity of the request cannot be validated it is discarded.

The proxy service uses ARM TrustZone's privileges to perform the requested memory operations. The proxy service's secure-world code has the permission to access both, the secure-world memory as well as the normal-world memory. Hence, the proxy service has access to D's entire memory. For read operations, the request contains the memory address to be read. The proxy service reads the content of the memory address, creates an integrity proof for the read value, authenticates it, and sends it back to the requesting entity.[7] For write operations the request contains the target memory address and the value to be written. The proxy service will write the value to the specified memory location. The remote entity can validate the correct execution of a write operation by performing a subsequent read operation.

The proxy service can enforce arbitrary access policies such as allowing and denying access to particular memory regions. For instance, it can restrict remote entities to access normal-world memory only.

Integrity and authenticity of remote requests as well as the responses can be assured either via asymmetric cryptography primitives or via symmetric cryptographic primitives, given a pre-established shared key.

5.2.2.2 *Regulating Devices in Restricted Spaces*

Subsequently we explain the individual steps for our solution and show that they fulfill the requirements described in Section 5.2.1.3.

Check-in. During check-in the host $\mathcal{H}$ wants to validate that a guest device $D_\mathcal{G}$ is benign and enforces $\mathcal{H}$'s policies. Our design accomplishes this in four steps.

Authentication. First, $\mathcal{H}$ and $D_\mathcal{G}$'s secure world, i. e., its TEE, have to mutually authenticate. Each $D_\mathcal{G}$ is equipped with a private/public key pair, which is only accessible to the secure world. The public key's authenticity is certified by the device manufacturer $\mathcal{M}$, allowing $\mathcal{H}$ to validate it. Similarly, $\mathcal{H}$ possesses an asymmetric key pair, with a certificate singed by a trusted entity assuring the authenticity of $\mathcal{H}$'s public key. Additionally, $\mathcal{H}$ has to ensure that the secure channel is established with the TEE of $D_\mathcal{G}$, i. e., that it is not subject to a cuckoo attack [300]. This can be achieved by employing the methods proposed by Zhang et al. [420] during the channel establishment.

Our design uses these authenticated keys to establish a secure channel between $\mathcal{H}$ and $D_\mathcal{G}$'s secure world, providing authenticity and integrity for all subsequent interaction.

Obtaining and Inspecting Guest State. After the mutual authentication $\mathcal{H}$ needs to (1) validate $D_\mathcal{G}$ state with the goal to gain trust that $D_\mathcal{G}$ will comply with its policies, and (2) learn the concrete configuration of $D_\mathcal{G}$.

First, $\mathcal{H}$ uses our remote memory operation primitive (cf. Section 5.2.2.1) to retrieve a memory snapshot of $D_\mathcal{G}$. In particular, the memory of the normal-world kernel is copied

7 If required the resulting message's confidentiality can be protected, as well.

and sent to the host. $\mathcal{H}$ analyses the received memory image of the normal-world kernel to detect if the kernel is compromised or if the kernel contains known vulnerabilities. $\mathcal{H}$ can, for instance, compare each kernel code page against a whitelist, e. g., reference code pages from approved Android distributions [248, 345]. Additionally, $\mathcal{H}$ can make sure that the kernel's data structures satisfy invariants that usually hold in uncompromised kernels [39] to detect Rootkits, which often modify crucial kernel data structures [310, 38, 312]. $\mathcal{H}$ can also utilize in-depth memory snapshot analysis for rootkit detection that was developed in prior works [310, 39, 81, 116, 194]. Furthermore, $\mathcal{H}$ can check that appropriate security mechanisms are active on $D_{\mathcal{G}}$, e. g., virus scanners or kernel protection mechanism such as (fine-grained) kernel address space randomization [162]. If $\mathcal{H}$ is satisfied with $D_{\mathcal{G}}$'s state, i. e., if $\mathcal{H}$ considers $D_{\mathcal{G}}$ secure and trusted, $\mathcal{H}$ analyzes the kernel image to identify installed peripherals.

Policy Enforcement. The host $\mathcal{H}$ of a restricted space defines a set of policies that all $\mathcal{G}$ (with their devices $D_{\mathcal{G}}$) need to comply to. In order to enforce these policies $\mathcal{H}$ remotely reconfigures all $D_{\mathcal{G}}$ entering the restricted space. In the previous step $\mathcal{H}$ receive an image of $D_{\mathcal{G}}$'s Operating System (OS) kernel and analyzed it to learn which peripherals are available on $D_{\mathcal{G}}$. $\mathcal{H}$ composes a list of memory modifications that are necessary to disable all peripherals that are not allowed to be available in its restricted space, e. g., the camera.

The usage of any peripheral device requires software routines that interact with it, i. e., a device driver. These drivers are typically well isolated as software modules with a defined interface to the kernel. To interact with the peripheral device (parts of) the driver needs to execute with kernel privileges. Therefore, all drivers are contained in the kernel image received by the host.

The host compiles a list of memory changes that will enforce its policies on $D_{\mathcal{G}}$ by modifying the kernel such that functionalities, which are not permitted in the restricted space, are unavailable. Specifically, the drivers of peripheral devices are disabled. This can be achieved either by overwriting the kernel's references (i. e., pointers) to the driver and its functions, or by replacing the driver functions with dummy functions. By using dummy functions, complex policies can be implemented. The dummy driver can return synthetic data or an error code to completely disable a peripheral device, or implement a subset of the peripheral's functionalities, e. g., enabling only voice calls for a cellular network modem while disabling data connections.

The list of all memory modifications is sent to $D_{\mathcal{G}}$. Using our remote memory operation primitive, the changes are applied to $D_{\mathcal{G}}$'s kernel. It makes sure that the kernel memory has not been changed in meanwhile, i. e., all memory to be modified contains the data assumed by the host when defining the memory modifications. If the modifications are applied successfully, $\mathcal{H}$ obtains a cryptographic hash, called verification token, covering all modified memory locations. Thus, $\mathcal{H}$ gets assurance that its policies are applied on $D_{\mathcal{G}}$. If the memory modifications are unsuccessful, $\mathcal{H}$ resorts to traditional measures, such as physically locking $D_{\mathcal{G}}$ away.

Policy Enforcement Persistence. Continuous checks are necessary to ensure that $\mathcal{H}$'s memory modification, which are necessary to comply to the restricted space's policies, are not reverted. $\mathcal{H}$ can – at any point in time while $D_{\mathcal{G}}$ is in the restricted space – request a

fresh verification token, i. e., a cryptographic hash of all memory locations that have been modified in order to enforce $\mathcal{H}$'s policies. To prevent reply attacks each request contains a unique nonce, which gets included in the verification token.

Because all modifications of $\mathcal{H}$ are applied to $D_{\mathcal{G}}$'s Random Access Memory (RAM), which is non-volatile memory, a shutdown and reboot of the device will revert them. While this will be detected by $\mathcal{H}$, the guest $\mathcal{G}$ might wish to shutdown $D_{\mathcal{G}}$ for benign reasons, for instance saving energy. We developed a solution that allows the suspension of $D_{\mathcal{G}}$ while preserving $\mathcal{H}$'s memory modifications. The details of our solution are described in our conference publication [64]

Check-out. During check-out, $\mathcal{H}$ needs to validate that $D_{\mathcal{G}}$ was not maliciously modified, e. g., to revert the memory modification $\mathcal{H}$ sent during check-in. $\mathcal{H}$ requests a new verification token from $D_{\mathcal{G}}$, which is compared against the verification token received during check-in (modulo the fresh nonce). If the validation token does not match $\mathcal{H}$'s expectation, i. e., the state has changes since the check-in process, $\mathcal{H}$ has to take additional measures, e. g., undertake a thorough inspection of $D_{\mathcal{G}}$.

Additionally, $D_{\mathcal{G}}$ must be restored to an unrestricted state. This can be achieved by restarting $D_{\mathcal{G}}$, i. e., resetting $D_{\mathcal{G}}$'s non-volatile RAM to a defined state. Alternatively, $\mathcal{H}$ can use our remote memory operation primitive to revert all memory changes it issued to $D_{\mathcal{G}}$. Using the memory image received during check-in, $\mathcal{H}$ can determine the original content of all memory locations that were modified to enforce $\mathcal{H}$'s policies.

5.2.3 *Implementation and Evaluation*

We implemented and evaluated the different components of our solution on an i.MX53 Quick Start Board from Freescale. The i.MX53 is a TrustZone-enabled development board equipped with a 1 GHz ARM Cortex A8 processor and 1 GB DDR3 RAM, which we used to implement and evaluate the secure-world components of our solution. In particular, we implemented and evaluated secure authenticated channel establishment, remote memory operations, verification token generation as well as our device suspension mechanism as described in our conference publication [64].

We further implemented a simple rootkit detection mechanism, which analyzes a kernel memory image for modified system-call handler functions, as a proof-of-concept for the host. Our evaluation shows that our remote memory operation primitive needs approximately 54 s to copy and authenticate all memory required for our rootkit detection, with the authentication via Keyed-Hash Message Authentication Code (HMAC) being the dominating factor.

To validate the feasibility of our kernel data modification approach and to evaluate its performance we developed a proof-of-concept implementation on a Samsung Galaxy Nexus smart phone. It is equipped with a Texas Instruments OMAP 4460 chipset (dual-core ARM Cortex-A9 processor at 1.2 GHz) and 1 GB of RAM and runs Android 4.3 based on the Linux kernel version 3.0.72. This platform provides more peripheral devices compared to the i.MX53 development board enabling more extensive evaluation of our approach. Our remote memory operation primitive is emulated by a kernel module

in the (normal world) OS as the secure world is not accessible on this smart phone. We implemented our approach for the Camera, WiFi, 3G modem (Voice and Data), Microphone, and Bluetooth, and show that all these peripheral devices can be successfully disabled by using our dummy driver approach.

Additionally, we implemented and evaluated a vetting service to verify the security of memory modifications needed to disable each of these peripherals.

We provide more details regarding our implementation and evaluation results in our conference publication [64].

5.2.4 *Conclusion*

Our policy enforcement mechanism, that we presented in this section, allows hosts to manage guest devices via remote memory operations. The host can analyze and regulate ARM TrustZone-enabled devices to enforce policies on such devices. Our solution requires only a minimal and simple trusted primitive on regulated devices. Utilizing ARM TrustZone, the host receives strong guarantees of guests' policy-compliance, while our vetting service allows guests to identify potential malicious policies by the host.

5.3 Related Work

In the following, we discuss works related to the trusted computing application presented in this chapter, which are based on Intel's Software Guard Extensions (SGX) and ARM TrustZone respectively. We first present solutions that aim at similar goals as our applications, i. e., privacy-preserving machine learning (in particular speech processing), mobile device management and (remote) memory operations. Afterwards we elaborate on selected applications that use the same Trusted Execution Environment (TEE) platforms as our solutions, i. e., Intel's SGX and ARM TrustZone, for different applications scenarios.

5.3.1 *Privacy-preserving Machine Learning*

In Secure Multiparty Computation (SMC), two or more parties execute an interactive cryptographic protocol to compute a function over private input data without revealing their inputs to each other or a third party. Using SMC privacy-preserving machine learning training [279], as well as classification [330, 254, 326, 92, 215], can be achieved.

Homomorphic Encryption (HE) enables processing of encrypted data, such that the decryption of the result is equal to the result of performing the same processing on the plain-text data. Microsoft's CryptoNets [135] was the first secure evaluation framework for neuronal networks based on HE, and was improved later with CryptoDL [188]. Despite these improvements, the reported evaluation results indicate that HE-based solutions are not suitable for real-time speech recognition tasks.

Pathak et al. [302] presented privacy-preserving speech processing, e. g., speech recognition and speaker verification, using SMC and HE techniques. The reported performance results (e. g., requiring more than 3 h computation time to recognize a single word from a 10 vocabulary for 1 s of audio) show the impracticality of this approach. Treiber et al. [380] improve the performance of speaker verification using SMC. However, solutions relying on cryptographic techniques incur significant overhead with regard to communication and computation cost.

Glackin et al. [163] proposed an approach that allows searching for keywords in encrypted outsourced speech documents. It utilizes searchable encryption on the server-side to deliver data corresponding to the phones of a keyword, which were extracted from the input audio by the trusted client beforehand.

Ohrimenko et al. [291] customized several machine learning algorithms, including neural networks, to run protected in an Intel SGX enclave. The algorithms were adapted in order to prevent cache-based side-channel attacks, allowing multiple parties to securely share their data, e. g., for joint training or evaluation of machine learning models. Chandra et al. [91] protect data analytics algorithms using SGX with high efficiency allowing real-time data processing. To thwart leakage through side channels, their approach introduces noise to memory traces observed by the adversary by accessing dummy data, rather than hiding memory access patterns. Chiron [198] proposes privacy-preserving machine learning as a service using SGX enclaves where the training algorithm is provided by one party and the (training) data by another party. Because the parties are mutually

distrusting the data-providing party cannot validate the algorithm executing in the enclave to ensure that it does not leak its data. Chiron solves this problem by providing two-way isolation.

Offline Model Guard (OMG) adapts the concept of VoiceGuard to enable protected speech recognition on user devices [42]. OMG leverages SANCTUARY (cf. Section 3.2) to isolate the speech recognition process on the user's device in order to protect the Intellectual Property (IP) of the model provider.

5.3.2 *Mobile Device Management*

Trends such as Bring Your Own Device (BYOD) have spawned a number of research projects as well as commercial solutions for Mobile Device Management (MDM), i. e., solutions enabling multiple persona [333, 272, 52] or enforcing access control policies on smart devices [401, 357, 77, 189].

Most of these solutions either require the smart device to run software that includes additional policy enforcement mechanisms, for instance, ASM [189] extends Android [172] with a set of security hooks to insert additional security policy checks, which can be installed via an app. Other approaches rely on virtualization of the guest device's software [22, 112, 393, 118], i. e., a trusted hypervisor provides isolation for individual Virtual Machines (VMs) for each persona.

While these approaches allow more complex policies one main benefit of our solution (cf. Section 5.2) lies in its simplified design and minimized complexity. Security-enhanced Operating Systems (OSs) and virtualization solutions require large and complex trusted policy-enforcement code to be executed on guest devices, in contrast, the Trusted Computing Base (TCB) of our solution is just a few thousand lines of code. Additionally, our solution provides security guarantees that are rooted in trusted hardware, and our policy enforcement engine is hardware-protected by ARM TrustZone. This enables our solution to provide the host with guarantees that the policy was enforce while the guest was in the restricted space, using our verification token concept (cf. Section 5.2.2).

Samsung Knox [36] and Sprobes [156] leverage TrustZone to increase the trustworthiness of the normal-world software by providing real-time protection against kernel-level rootkits. Operations of the normal-world OS kernel that manipulate critical kernel data are outsourced to the secure world, where additional checks are performed before applying the changes in order to prevent rootkits. Knox's real-time kernel protection is orthogonal to our solution; however, our solution can benefit from the additional security Knox and similar systems can provide.

TrUbi [108] provides a framework to enforce usage restriction, e. g., disable the use of network functionalities or the microphone, on Android devices using Mandatory Access Control (MAC), leveraging ASM [189]. This approach entrusts the enforcement of the restriction policies to the Android kernel resulting in a large and complex TCB. Our work aims to provide strong security for the TCB by protecting it with ARM TrustZone while reducing the size and complexity of the TCB by limiting it to minimal functionalities, i. e., remote memory operations.

Ditio [275] utilizes a hybrid architecture using hardware virtualization and ARM TrustZone to provide auditing for sensor usage on mobile and smart home devices. In contrast to our solution, Ditio does not enforce any sensor usage policies, it rather records sensor activities to allow detection of devices' misbehavior after the fact. To be able to record all sensor activities, Ditio relies on a trusted hypervisor that virtualized all sensors and logs their activities, occupying the hardware virtualization feature and preventing its usage for other purposes. Furthermore, the hypervisor incurs additional memory and computational cost, also during normal operation of the device.

SeCloak [240] uses TrustZone to control the on/off state of peripheral devices on ARM devices by assigning them to and controlling them from the secure world. In order to minimize the TCB, SeCloak emulates the devices such that normal-world driver can be used to operate them while giving the secure world ultimately control over the device. Our solution allows more fine-grain control over peripherals, e. g., restricting a cellular modem's use for data transfer while allowing voice calls.

Vijeev et al. [391] consider restricted spaces for drones and propose a policy enforcement scheme for drones. Their scheme aims at controlling the usage of data rather than the generation of data, i. e., instead of controlling whether a peripheral device is allowed to be used their scheme controls how (by which software module) the generated data is used. While this enables new usage scenarios it also increases the TCB as it requires trusted information-flow and policy enforcement mechanisms on the device.

5.3.3 *(Remote) Memory Operations*

TrustDump [370] utilizes TrustZone to reliably acquire memory pages from the normal world. Solutions such as Android LiME [186] and others [373, 366] enable memory acquisition independent of ARM TrustZone. While these solutions provide remote read operations, similar to our solutions, they do not provide the means to enforce policies. TrustDump, for example, focuses on memory introspection to detect malware.

The Ninja framework for malware analysis uses ARM TrustZone to transparently trace the execution of software on ARM devices [285]. The goal of Ninja is to record the behavior of software executed in the normal world, while our work allows to modify the normal-world software using remote memory operations.

SATIN proposes a TrustZone-based asynchronous introspection mechanism for multi-core ARM systems [395], which prevents that an adversary controlling the normal world can evade detection from the secure world be utilizing the asynchronous execution of the secure world on one CPU core while the other CPU core continue executing adversary-controlled normal-world code.

Hardware Interfaces. Remote memory operations are used, for instance, in data centers where Remote Direct Memory Access (RDMA) interfaces are used to bypass the performance overhead of general purpose network protocols such as TCP/IP in distributed systems [271, 377]. While these interfaces have been repurposed, e. g., for kernel malware detection [311] and remote repair [55], they rely on a PCI-based co-processor on a device via which another device can remotely transfer and modify memory pages.

On personal devices, the IEEE 1394 (Firewire) interface allows a connected peripheral device to directly access memory. However, this interface is typically not available on smart devices. Hardware debugging and testing interfaces, such as the widely used Joint Test Action Group (JTAG) interface [213], also allow read/write access to a device's memory as well as Central Processing Unit (CPU) registers. However, JTAG is not easily accessible on consumer devices and might be disabled by the device manufacturer.

5.3.4 *TEE Applications*

ARM TrustZone. Microsoft's TLR [335] and Nokia's ObC [234] utilize TrustZone to protect user app in isolated executions environments, such that an adversary with high privileges, i. e., an adversary that has compromised the OS kernel, cannot manipulate these protected apps or extract sensitive data from them. Liu et al. [253] use TrustZone to ensure the trustworthiness of sensor readings from peripheral devices. Various mobile payment systems, for instance Samsung Pay [334], rely on TrustZone to secure payment processes. Further applications of TrustZone include mobile data billing [320], attesting mobile advertisements [242], and implementing the TPM-2.0 specification in firmware [321].

Intel SGX. Using TEEs, in particular Intel SGX, to enable SMC has been suggested in different works [227, 238], including our own work [315]. This general concept enables the use of TEEs for various use cases and applications.

VC3 adapts the MapReduce computing paradigm for distributed computing to Intel SGX [339]. Mapper and Reducer nodes are isolated in SGX enclaves. However, VC3 does not prevent the leakage of sensitive information from the (encrypted) data flows between mapper and reducer enclaves.

Haven allows the isolation of unmodified applications inside an SGX enclave [41]. It utilizes a library OS to provide proxy system calls from the application to the untrusted OS outside the enclave. The authors show that Haven can be used to isolate an entire Database Management System (DBMS) in an enclave. However, Haven does not consider information leakage due to side channels or access patterns to external resources. Furthermore, SGX's memory limitation impacts the performance of Haven for large and complex applications.

6 SECURITY SERVICE: REMOTE ATTESTATION

Security services provide the functional building blocks that are required to build secure systems. In this chapter, the focus is on Remote Attestation (RA), a security service that enables one entity, called *verifier*, to check the integrity of another (remote) entity's internal state, called *prover* (see Section 2.3.2 for details).

RA is particularly relevant for small embedded systems, which often do not have the resource for sophisticated, and therefore complex, defense mechanisms. The security of embedded devices is a timely and important issue, due to the proliferation of these devices into numerous and diverse settings. One contributing factor is the constantly increasing integration and introduction of such devices into many spheres of everyday life, including: automotive, avionics, factory automation, household, medical, and public utilities. At the same time, growing presence of computerized components in previously non-electronic (mechanical or simply analog) objects and tasks represents an attractive set of new and exciting attack surfaces for nefarious individuals and organizations.

While RA has been studied extensively in the past, this work sheds light on new aspects of RA and develops new concepts for attestation in collaborative systems.

Most attestation schemes naturally focus on the scenario where the verifier is trusted and the prover is not. The opposite setting – where the prover is benign, and the verifier is malicious – has been side-stepped. The prover can be affected by attack scenarios such as verifier impersonation, Denial-of-Service (DoS) and replay attacks, all of which result in unauthorized invocation of attestation functionality on the prover. We argue that protection of the prover from these attacks must be treated as an important component of any RA method. Section 6.1 addresses the issue of prover security.

Another restriction of previous Remote Attestation schemes is their limitation to single device scenarios, i. e., they are concerned with the verification of a *single* prover device. SEDA presents the first attestation scheme for device swarms (Section 6.2), allowing the efficient attestation of systems composed of multiple – possibly collaborating – devices.

However, with SEDA all devices attest to a single external verifier. DIAT enables devices *within* autonomous collaborative networks to interact with each other securely and efficiently, relying on a minimal Trusted Computing Base (TCB) (Section 6.3). It presents a novel approach that allows to verify the correctness of data exchanged by collaborating devices, rather than verifying the correctness of entire device, by attesting the correct generation and processing of critical data using control-flow attestation.

6.1 Remote Attestation for Low-End Embedded Devices: the Prover's Perspective

The research community recognized the danger posed by insecure embedded devices and responded with the development of countermeasures. Remote Attestation (RA) is one particularly important countermeasure for embedded device, which have limited resources to implement defense mechanism that are common in personal computers or servers. However, all attestation approaches proposed before the publication of our paper [65] involve an interactive protocol between a trusted verifier and a potentially compromised prover. Although putting the focus on this setting is both natural and sensible, it has overshadowed another important issue – attacks on the prover that can be launched through the attestation protocol itself. As we will elaborate in this section, attacks that misuse the RA service by maliciously invoking attestation functionality on a prover device pose a real threat. Therefore, any comprehensive attestation method must include means to prevent or resist them. Attacks leveraging verifier impersonation (e. g., via replay, reorder, or delay of attestation requests) are particularly dangerous, since they are not trivially prevented and amount to an effective Denial-of-Service (DoS) attack. Such attacks can exhaust energy (deplete batteries) and prevent the targeted device from performing its primary – possibly critical – tasks, such as control, sensing, or actuation.

Goals and Contributions. To goal of this work is to extend RA in order to secure it against strong adversaries that misuse a prover device's attestation capabilities to launch DoS attacks on the prover.

We identify and analyze DoS attacks leveraging RA to target low-end embedded prover devices. First, we extend attestation protocols with well-known cryptographic techniques to secure them against a simple external adversary. Then, we investigate a more sophisticated *roaming* adversary that can temporarily compromise the prover device to overcome our first extensions. The roaming adversary compromises the prover and manipulates it in a way that is undetectable by standard attestation methods. These attacks are particularly dangerous since the roaming adversary can remain stealthy by erasing all traces of its presence. We demonstrate how the roaming adversary can be mitigated by extended attestation techniques for low-end embedded systems with minimal hardware assumptions. We provide two implementations, a base version implemented through limited hardware extensions, and an advanced version that does not require hardware component beyond what is commonly available in low-end Microcontroller Units (MCUs). We believe that countermeasures developed in this work represent a significant improvement and an advantage over previously established attestation techniques.

6.1.1 *System and Adversary Model*

As introduced in Section 2.3.2, an attestation protocol is an interaction between a prover ($\mathcal{PRV}$) and a verifier ($\mathcal{VRF}$). $\mathcal{VRF}$ needs to determine whether $\mathcal{PRV}$ is in a known good (and therefore trusted) state. $\mathcal{VRF}$ invokes the attestation protocol by sending a request (attreq) to $\mathcal{PRV}$. We assume that $\mathcal{PRV}$ has a trust anchor responsible for measuring $\mathcal{PRV}$'s

state and sending the result back to $\mathcal{VRF}$. Further, we assume $\mathcal{PRV}$ and $\mathcal{VRF}$ share a key (K_{Attest}), which is exclusively accessible by $\mathcal{PRV}$'s trust anchor and is used to authenticate the attestation result.

6.1.1.1 *Attestation as Denial-of-Service*

RA techniques typically assume that $\mathcal{VRF}$ is trusted while $\mathcal{PRV}$ is not trusted and must be verified. However, a prover has no assurance whether it is interacting with a *real verifier*. Without authentication of the attestation request attreq, an adversary can trivially impersonate the verifier by sending bogus attestation requests to a prover. Since $\mathcal{PRV}$ has no means to distinguish whether an attestation request is genuine or not, $\mathcal{PRV}$ invokes its local attestation functionality in any case. Executing the attestation routine results in a waste of energy (by depleting batteries) and takes $\mathcal{PRV}$ away from performing its primary tasks, such as control, sensing, or actuation.

If an adversary impersonates $\mathcal{VRF}$ and initiates the attestation routine of $\mathcal{PRV}$, the following operations have to be performed by $\mathcal{PRV}$, incurring non-negligible cost. $\mathcal{PRV}$ has to generate an attestation response reflecting $\mathcal{PRV}$'s state, this typically involves computing a Message Authentication Code (MAC) over the prover's entire writable memory. There are two common approaches to implement a MAC: (1) a CBC-based function with a block cipher (such as Advanced Encryption Standard (AES)) or (2) a keyed hash function (such as SHA1-Keyed-Hash Message Authentication Code (HMAC) [237]). To illustrate MAC costs, Table 10 shows the time for computing a SHA1-HMAC on variable input sizes, using the *Intel Siskiyou Peak* embedded processor as the hardware platform [324]. Computing a SHA1-Hash over its 512 kB of Random Access Memory (RAM) takes $\approx$ 754 ms. The RAM is processed in blocks of 64 Byte, i. e., $(512\,\text{kB}/64\,\text{Byte}) = 8192\,\text{blocks}$. Processing each block takes ≈ 0.092 ms (cf. Table 10), plus an additional fixed cost for the initialization and finalization of the hash computation results in overall cost of $(512\,\text{kB}/64\,\text{Byte}) \cdot 0.092\,\text{ms} + 0.340\,\text{ms} \approx 754\,\text{ms}$.

Even worse, current low-end device attestation techniques assume that the attestation routine runs without interruption [139, 368]. Thus, gratuitous (malicious) invocation of attestation can be detrimental to the execution of prover's main – possibly critical – functions. Our real-time complaint security architecture TyTAN (cf. Section 3.1) does not fundamentally address this problem, as it (1) relies on a managing software, e. g., an Operating System (OS), which is not available on all low-end embedded systems, and (2) the prover device is still affected by the attestation request, e. g., leading to increased energy consumption.

The core reason for the ease of DoS attacks based on RA is that the prover's workload is significantly higher than that of the verifier. This asymmetry is not limited to the sheer *amount* of work performed by each party; it also occurs due to the fact that the verifier is generally much more powerful than the prover, which might be a low-end MCU.

SHA1-HMAC [237]		AES-128 (CBC)		
			per block	
Fix	per block	Key exp.	Enc	Dec
0.340 ms	0.092 ms	0.074 ms	0.288 ms	0.570 ms

Speck 64/128 (CBC)			ECC (secp160r1)	
	per block			
Key exp.	Enc	Dec	Sign	Verify
0.016 ms	0.017 ms	0.015 ms	183.464 ms	170.907 ms

Table 10: Performance of cryptographic primitives on Intel Siskiyou Peak at 24 MHz.

6.1.1.2 *Adversaries*

We define two types of adversaries envisaged in the context of verifier impersonation and DoS attacks on the prover by misusing RA. Neither type is capable of physical attacks, and we assume the prover's Trusted Computing Base (TCB) to be immune against attacks.

External Adversary $\mathcal{A}dv_{ext}$. We first consider an *external* adversary ($\mathcal{A}dv_{ext}$) that can control all communication between $\mathcal{PRV}$ and $\mathcal{VRF}$. $\mathcal{A}dv_{ext}$ can drop, insert and delay messages, following the well-known Dolev-Yao model [133]. However, being strictly external, $\mathcal{A}dv_{ext}$ cannot directly manipulate any internal state of $\mathcal{PRV}$.

Roaming Adversary $\mathcal{A}dv_{roam}$. A stronger and more sophisticated adversary is the *roaming* adversary ($\mathcal{A}dv_{roam}$). It can, in addition to $\mathcal{A}dv_{ext}$'s capabilities, infect $\mathcal{PRV}$ with malware and later *cover its tracks* by erasing its malware. Similar to $\mathcal{A}dv_{ext}$, we assume that $\mathcal{A}dv_{roam}$'s primary goal is to perform a DoS attack on $\mathcal{PRV}$. $\mathcal{A}dv_{roam}$ operates in three phases:

- Phase I: $\mathcal{A}dv_{roam}$ eavesdrops on and collect genuine attestation request sent from $\mathcal{VRF}$ to $\mathcal{PRV}$.
- Phase II: $\mathcal{A}dv_{roam}$ compromises $\mathcal{PRV}$ (e. g., via malware), changes local state, and leaves $\mathcal{PRV}$, i. e., erases all traces of its presence.
- Phase III: $\mathcal{A}dv_{roam}$ replays previously recorded attestation requests.

Note that, in Phase II, $\mathcal{A}dv_{roam}$ only changes dynamic data on $\mathcal{PRV}$. This is not detectable by subsequent attestations, which only cover the code and static data of $\mathcal{PRV}$. Additionally, $\mathcal{A}dv_{roam}$ can extract other information from $\mathcal{PRV}$, e. g., an authentication key K_{Attest}.

In the following section we discuss mitigation techniques against these adversaries. In the process, we also identify the requirements these mitigation techniques pose on underlying attestation protocols and the prover device's hardware features.

6.1.2 *Mitigating* $\mathcal{A}dv_{ext}$

We start with mitigation strategies to fend off $\mathcal{A}dv_{ext}$. Preventing attacks by an external adversary requires (1) authentication for attestation requests, and (2) countermeasures against replay (and related) attacks.

6.1.2.1 *Authenticating Verifier Requests*

It is quite evident that, in order to mitigate bogus attestation requests (or, equivalently, verifier impersonation attacks), a verifier must authenticate itself to the prover. The authentication, in a remote scenario, can be done via either public or symmetric key cryptography. With the former, $\mathcal{VRF}$ signs its attestation request and the prover validates the request's authenticity with the verifier's public key. $\mathcal{PRV}$ needs assurance about the authenticity of $\mathcal{VRF}$'s public key, e. g., it can be stored in the prover's non-malleable memory. With symmetric cryptography, $\mathcal{VRF}$ and $\mathcal{PRV}$ are assumed to share a secret key. The verifier appends a MAC to the attestation request and the prover validates the MAC by recomputing the same on its side, thus authenticating the request.

As can be expected, public key cryptography is expensive for low-end MCUs. Table 10 lists the cost of different cryptographic primitives that can be used for authentication. It shows that even relatively efficient Elliptic Curve Cryptography (ECC) incurs significant computational cost for the prover (170 ms). Thus, with the use of ECC, simply authenticating verifier's attestation request can occupy the prover devices such that an adversary can misuse requests with invalid signatures to launch a DoS attack. This leads to the paradoxical situation where a mechanism meant to prevent DoS attacks itself can result in a DoS attack. Consequently, we conclude that the use of public key cryptography is not suitable in this context.

Using symmetric cryptography to secure authenticated attestation requests yields significantly better performance: a SHA1-based HMAC can be validated in 0.432 ms. Assuming an authentication request concatenated with the key is small enough to fit into a single SHA1-HMAC block, the required computation time is $0.340\,\text{ms} + 0.092\,\text{ms} = 0.432\,\text{ms}$. Standard block ciphers such as AES perform slightly better, requiring $0.074\,\text{ms} + 0.288\,\text{ms} = 0.362\,\text{ms}$ for key expansion and the encryption of one block. Using lightweight block ciphers such as Speck [43] further reduces the cost by a factor of ten: $0.016\,\text{ms} + 0.017\,\text{ms} = 0.033\,\text{ms}$. If key expansion is done in advance, the performance of the block ciphers can be further improved. Messages are assumed to fit into one block for each cryptographic primitive: ECC: 160 bit, AES: 256 bit, Speck: 64 bit, and HMAC: 512 bit.

Requirements. The use of symmetric cryptography to compute a MAC imposes the requirement to protect the authentication key in an access-restricted hardware-protected key storage (e. g., in Read-Only Memory (ROM)). This requirement exists already for recent attestation architectures such as SMART [139], SPM [368] and TrustLite [226]. They have the same requirements for the prover to be able to compute an authenticated response, i. e., a challenge-based MAC over the prover's memory.

6.1.2.2 *Handling Replay, Reorder & Delay*

Unfortunately, mere authentication of attestation requests is not sufficient to mitigate DoS attacks. $\mathcal{A}dv_{ext}$ can simply eavesdrop on and record genuine attestation requests and later replay them. Alternatively, $\mathcal{A}dv_{ext}$ can intercept and arbitrarily delay or reorder genuine requests. There are several standard ways to detect replay, reordering and delay attacks:

- *Nonces:* If each attestation request includes a nonce (i. e., a unique value) provided by the verifier, the prover can keep a complete nonce history of previously received (and authenticated) attestation requests. This allows $\mathcal{PRV}$ to detect replayed request, i. e., requests with a previously seen nonce.
- *Counters:* If each attestation request includes a monotonically increasing counter, the prover accepts a new request only if its counter is strictly greater than the last counter value received and processed. The new counter then replaces the previous one. Requests bearing out-of-order or duplicate counters are rejected.
- *Timestamps:* Assuming synchronized clocks between both parties and sufficiently inter-spaced genuine attestation requests, allows the prover to detect replayed, reordered and delayed messages. The verifier includes an up-to-date timestamp in the attestation request and the prover only accepts request with a timestamp close to the current time.

Using nonces is problematic for two reasons: (1) It poses high demands on the availability of non-volatile memory on the device to store a complete nonce history. (2) Nonces protect only against replayed requests, while reordered or delayed requests cannot be detected. Therefore, we consider nonces not suitable for our scenario, and rule them out for the remainder of this work.

For the counter-based approach the prover is required to maintain a sequence number, which also has to be stored in non-volatile memory. However, only the latest counter value has to be stores, hence, only a small and fixed amount of memory is required. Under the assumption that non-volatile memory is already available on the prover's platform – a feature available on many MCUs – no new architectural features are required compared to those identified by ElDefrawy et al. [139] for RA, thus enabling secure Remote Attestation (RA) in the untrusted-verifier model at minimal cost. On the other hand, a counter does not protect against delayed request attacks.

Timestamps offer the best security, under aforementioned assumptions on the adversary's capabilities. They protect against attacks leveraging replay, reorder or delayed requests. However, the major requirement imposed by timestamps is availability of a reliable real-time clock on the prover – a feature not previously identified as necessary for attestation.

Table 11 summarizes security features attainable with each approach.

Requirements. To protect against replay and reorder attacks, non-volatile memory is required. Many MCUs provide non-volatile memory, thus, minimal additional cost

	Feature:		
Attack:	**Nonces**	**Counter**	**Timestamps**
Replay	✓	✓	✓
Reorder	-	✓	✓
Delay	-	-	✓

Table 11: Summary of DoS attack mitigation features.

is imposed by this protection mechanisms. Detection of delayed requests requires a synchronized real-time clock on the prover.

6.1.3 *Mitigating* $\mathcal{A}dv_{roam}$

The countermeasures discussed above that are based on counters and real-time clocks can easily be defeated by $\mathcal{A}dv_{roam}$, as we will explain below. We then develop mitigation techniques that can protect against $\mathcal{A}dv_{roam}$.

- $\mathcal{A}dv_{roam}$ *and Counters:* We assume an attestation mechanism that is secure against replay and reorder attack from an exteranl adversary ($\mathcal{A}dv_{ext}$), i. e., (1) each attestation request contains a monotonically increasing counter, and (2) the prover stores the counter from the last genuine attestation request in non-volatile memory. Without loss of generality, we assume that the adversary wants to replay a single attestation request. In Phase I, $\mathcal{A}dv_{roam}$ records one genuine attestation request $\mathsf{attreq}(i)$ where i denotes the counter. In Phase II, $\mathcal{A}dv_{roam}$ modifies the counter stored by the prover from i to $i-1$. It then leaves the prover, and after waiting arbitrary length of time, replays $\mathsf{attreq}(i)$. After checking its stored (modified) last counter, the prover accepts $\mathsf{attreq}(i)$ as fresh and performs attestation. The prover's counter is changed to i.

- $\mathcal{A}dv_{roam}$ *and Timestamps:* We assume an attestation mechanism that protects replay, reorder and delay attacks from an external adversary ($\mathcal{A}dv_{ext}$), i. e., (1) each attestation request is timestamped by the verifier, (2) the prover has a clock, and (3) prover's and verifier's clocks are synchronized. Again, we also assume that, in Phase I (see Section 6.1.1.2), $\mathcal{A}dv_{roam}$ records one genuine attestation request $\mathsf{attreq}(t_i)$ where t_i denotes the timestamp. In Phase II, $\mathcal{A}dv_{roam}$ re-sets the prover's clock to time $t_i - \delta$. It then leaves the prover, and after waiting for δ time units, replays $\mathsf{attreq}(t_i)$. After consulting its (modified) clock, the prover accepts it as timely and performs attestation. The prover's clock remains behind.

Although the DoS attack succeeds in both cases, there are two subtle differences: First, $\mathcal{A}dv_{roam}$ is more constrained with timestamps since it is bound to δ wait time before replay in Phase III. Second, resetting the prover's clock in Phase II leaves some evidence of the attack since the prover's clock remains behind. In contrast, resetting the counter allows $\mathcal{A}dv_{roam}$ to bring the prover back to its expected state. This means that the DoS attack is undetectable after the fact.

Protecting Key, Counters & Clocks. In Phase II, $\mathcal{A}dv_{roam}$ compromises the prover. At this time, it controls the prover device and can take actions to prepare for the actual DoS attack in subsequent Phase III. For example, $\mathcal{A}dv_{roam}$ could extract $\mathcal{PRV}$'s K_{Attest}, which would allow it to generate authentic attreq-s. Hence, K_{Attest} must be protected from read access. Only the trusted attestation code $Code_{Attest}$ must be allowed to read K_{Attest} (see Figure 32). Note that this is impossible in software-based attestation (see Section 6.4.1). Similarly, K_{Attest} must be write-protected; otherwise, $\mathcal{A}dv_{roam}$ could overwrite it with any key it chooses and achieve the same result. The counter in the last authentic (processed) attreq as well as $\mathcal{PRV}$'s local clock state must made immutable for $\mathcal{A}dv_{roam}$, in order to prevent replay, delay and reorder attacks described in Section 6.1.2.2. At the same time, the stored counter value must be updated with every new attestation request. Hence, the counter must be writable only by the prover's TCB, i. e., $Code_{Attest}$.

Requirements. In summary, to protect against $\mathcal{A}dv_{roam}$, K_{Attest} must be kept confidential and non-malleable, i. e., read-only and readable only by $Code_{Attest}$. The counter and the clock must be write-protected, they do not require confidentiality. The required protections can be achieved with minimal hardware security mechanisms already available in existing attestation architectures. In particular, low-end device attestation techniques, such as SMART [139] or TrustLite [226], already rely on hardware features and extensions to protect $Code_{Attest}$ and K_{Attest} against software attacks. These existing mechanisms can be used to protect a counter as well as a clock, as described in the next section.

6.1.4 *Implementation*

We now describe our prototype implementations of our proposed countermeasures. First, we describe the concept of Execution-Aware Memory Access Control (EA-MAC), a hardware protection mechanism used by multiple attestation solutions [139, 226, 62]. Then, we describe how we can utilize this mechanism to extend previous attestation techniques to protect against $\mathcal{A}dv_{roam}$.

6.1.4.1 *Execution-Aware Memory Access Control*

Execution-Aware Memory Access Control (EA-MAC) is a primitive that allows to restrict access to critical components of the system (i. e., K_{Attest}, the counter and the clock). It has been used in several prior proposals [139, 226], and by our embedded security architecture TyTAN (cf. Section 3.1). The core idea of EA-MAC is to allow or denial read and write access to memory depending on currently executing code, i. e., memory locations are only accessible for selected software components (see Section 2.1.1 for a more detailed description). For example, access to K_{Attest} is limited such that only $Code_{Attest}$ can read it. Hence, even in the case that all code (except $Code_{Attest}$) is compromised, K_{Attest} remains protected. To protect $Code_{Attest}$ itself from being maliciously modified it has to be non-malleable, e. g., in SMART [139] it resides in ROM. TrustLite [226] and TyTAN [62] use secure boot and EA-MAC-based isolation to maintain the integrity of $Code_{Attest}$.

The basic operation of EA-MAC is approximately the same in all attestation architectures: The Central Processing Unit (CPU) allows a particular memory access based on the memory location of the currently executing code. The currently executing code's memory location can be retrieved from the value of the Program Counter (PC). However, the individual implementations differ in some aspects. For instance, SMART [139] has a single memory element (K_{Attest}), which is protected by a hard-wired EA-MAC. In contrast, TrustLite [226] allows flexible configuration of protected memory regions and associated access policies at run time by software. An extended Memory Protection Unit (MPU), called Execution-Aware Memory Protection Unit (EA-MPU), specifies which code region has access to which data region. Notably, the EA-MPU can control access to arbitrary memory regions, i. e., access to memory-mapped configuration registers of peripheral devices can be controlled, as well.

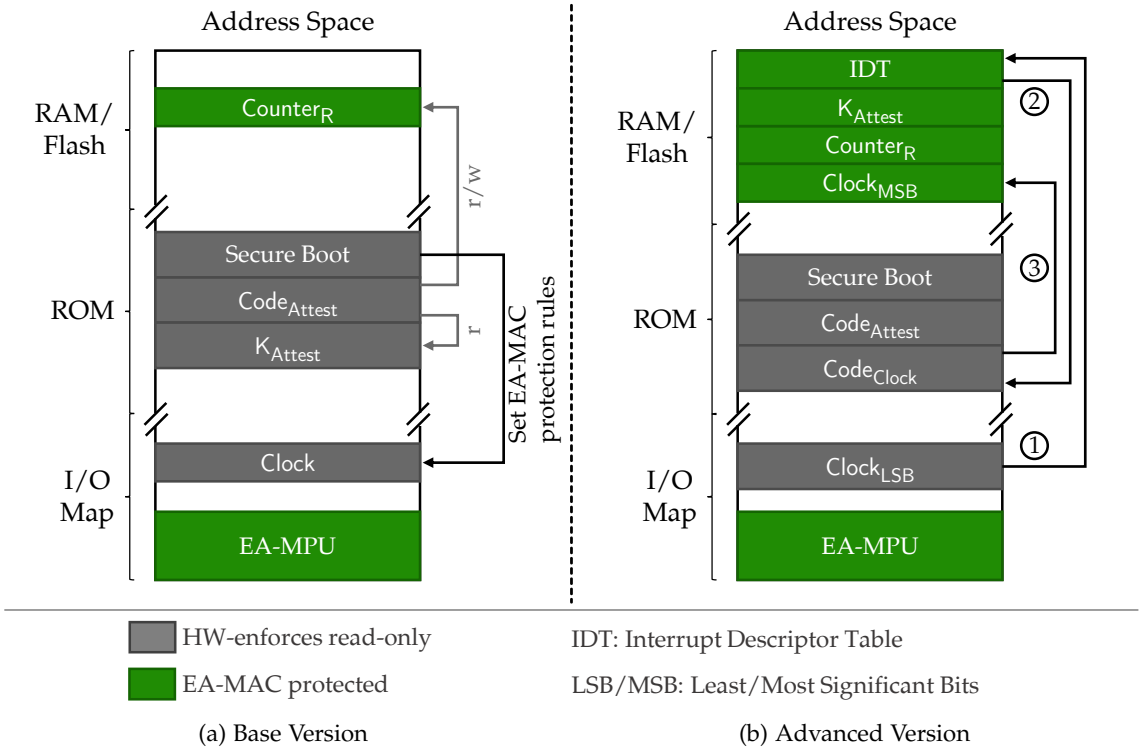
6.1.4.2 *Implementation Details*

The prototype implementation of our $\mathcal{A}dv_{roam}$ countermeasures is based on three components: ROM, EA-MAC and secure boot. We developed two versions of our prototype as shown in Figure 32. The basic version (Figure 32a) utilizes a real-time clock in hardware. Our more advanced version (Figure 32b) requires no new hardware features at the cost of a more complex software implementation. Both versions are based on the TrustLite attestation architecture. However, the same countermeasures are easily adaptable to other attestation techniques, such as SMART [139] as well as our security architecture TyTAN [62].

As discussed earlier, to mitigate $\mathcal{A}dv_{roam}$, three components must be protected: K_{Attest}, the counter, and the real-time clock.

Secure Boot. Protection of critical system components is realized by setting appropriate memory access rules in the EA-MPU. However, if the adversary controls the privileged systems software, such as the OS, it could change these rules and disable protection. For this reason, the system is started via secure boot, i. e., at boot time it verifies that correct software is loaded. This initial software sets up memory protection rules in the EA-MPU and locks it, by setting the EA-MPU configuration immutable, to prevent further changes, even from privileged software This can be done by the EA-MPU itself, via memory-mapped configuration registers. Setting the EA-MPUs configuration registers as read-only, protects EA-MPU rules from being changed, as shown in Figure 32a.

Keys & Counters. The secret attestation key K_{Attest} must be both read- and write-protected. By storing K_{Attest} in ROM, it is inherently write-protected. If it is stored in writable memory (e. g., RAM or Flash), it must be write-protected by a dedicated EA-MPU rule. In both cases, an EA-MPU rule is required to ensure that only $Code_{Attest}$ can read K_{Attest}. Hence, $\mathcal{A}dv_{roam}$ controlling all software (except $Code_{Attest}$) can neither read nor write K_{Attest}. At the same time, $\mathcal{A}dv_{roam}$ cannot modify $Code_{Attest}$, which is write-protected (in ROM). To protect $Code_{Attest}$ against run-time attacks, known mitigation techniques, e. g., limiting code entry points, and using Control-Flow Integrity (CFI) [123], can be utilized. Similarly, `ctr` is protected by an EA-MPU rule that makes it writable only by $Code_{Attest}$.



(a) Base Version (b) Advanced Version

Figure 32: (a) Base version of $\mathcal{A}dv_{roam}$ mitigation; K_{Attest} and ctr are only accessible by $Code_{Attest}$. Access control is enforced by an EA-MPU that is set up at system start by a secure boot mechanism. (b) Advanced version for a common low-end MCU clock design; $Clock_{LSB}$ is a short-term counter, which issues an interrupt when it wraps around ①. The immutable interrupt handling engine ensures that $Code_{Clock}$ serves the interrupt ②; $Code_{Clock}$ maintains a software counter ($Clock_{MSB}$) such that $Clock_{MSB} + Clock_{LSB}$ form a real-time clock ③.

Real-Time Clock. A real-time clock is needed if protection against delay attacks is required. Recall that a counter is sufficient to mitigate replay and reorder attacks. Obviously, the clock must be write-protected to prevent $\mathcal{A}dv_{roam}$ from setting it to an earlier value. In the simplest case, the clock counter is sufficiently large in size, i. e., in terms of number of bits, to prevent that it wraps around within the expected lifetime of the prover. Our base version, as shown in Figure 32a, can be used if such a clock counter is available. We evaluate the expected lifetime of prover device dependent on the clock counter's size in Section 6.1.5.

If a sufficiently large clock counter is not available then our second, more advanced version can be used. Figure 32b shows a platform with a short-term counter ($Clock_{LSB}$) which issues an interrupt when it wraps around. The interrupt is handled by the trusted and integrity-protected $Code_{Clock}$. This code maintains higher-order bits of the clock in writable memory ($Clock_{MSB}$). Again, the hardware counter must be read-only and $Clock_{MSB}$ memory must be protected with an EA-MPU rule to ensure that it is exclusively writable by $Code_{Clock}$. Additionally, the system's configuration of interrupt handling must

be protected. For example, if $\mathcal{A}dv_{roam}$ manipulates the Interrupt Descriptor Table (IDT), it could prevent $\mathsf{Code}_{\mathsf{Clock}}$ being invoked upon a wrap-around of $\mathsf{Clock}_{\mathsf{LSB}}$, thus effectively stopping the real-time clock. To prevent this, IDT's configuration can be locked down similar to the EA-MPU's configuration, by setting the memory region containing the IDT read-only during secure boot. Further, depending on the underlying MCU platform, disabling the timer interrupt must be prevented, and the location of the IDT itself must be immutable.

6.1.5 *Evaluation*

In this section, we evaluate the costs of our prototype implementations associated with our proposed countermeasures. The cost for protecting $\mathsf{K}_{\mathsf{Attest}}$ and ctr is the same in all variants, i. e., for each one EA-MPU policy must be configured, hence, one additional configuration slot in the EA-MPU is required. This is independent of the storage location of $\mathsf{K}_{\mathsf{Attest}}$, i. e., whether $\mathsf{K}_{\mathsf{Attest}}$ is stored in ROM or in mutable memory such as RAM or Flash. In both cases the cost is identical: Both, setting $\mathsf{K}_{\mathsf{Attest}}$ read protection (in ROM) as well as making $\mathsf{K}_{\mathsf{Attest}}$ read/write protection in RAM or Flash requires a single EA-MPU rule.

Clock Implementations. We evaluated our two variants for implementing a real-time clock. The first variant depends on a dedicated counter register that does not wrap around within the lifetime of the prover. For example, a 64 bit register incremented every clock cycle wraps around after 24 372.6 $years$ on a 24 MHz CPU. Table 12 shows the hardware cost for a counter register of this size as well as combinational logic for incrementing it. By reducing the clock frequency, the register size of the clock counter register can be decreased while preventing a wrap-around of the counter with realistic assumptions on the lifetime of a device. For example, given a 32 bit register, the wrap-around time is approximately 3 min at 24 MHz. By dividing the clock by $2^{20} = 1\,048\,576$ (i. e., incrementing it every one millionth cycle), a wraparound will occur only after 6 $years$ while keeping clock resolution at 42 ms.

Our second implementation is based on a clock design present in Intel Siskiyou Peak and other popular low-end MCUs, e. g., Texas Instruments MSP430 [161]. Reusing the existing clock induces no additional hardware cost. However, to protect our implementation for short-term counters incurs cost due two factors. (1) Protecting interrupt handling, and (2) storage of the software-maintained fraction of the current clock value ($\mathsf{Clock}_{\mathsf{MSB}}$). Table 12 shows the cost of this "software clock" (SW-clock), which consists of two EA-MPU protection rules: The first EA-MPU rule is required to set the IDT immutable; the second EA-MPU rule enforces protection of the memory location where the high-order bits of the clock value, $\mathsf{Clock}_{\mathsf{MSB}}$ in Figure 32b, are stored.

Overhead. We compare our implementations against a base-line system that supports attestation without protection against $\mathcal{A}dv_{ext}$ or $\mathcal{A}dv_{roam}$. The base-line system requires an EA-MPU with at least two rules: The first rule is needed to protect the EA-MPU itself from modifications at run time. The second rule protects $\mathsf{K}_{\mathsf{Attest}}$. The total cost of the base-line system is $5528 + 278 + (116 \cdot 2) = 6038$ registers and $14361 + 417 + (182 \cdot 2) =$

	Siskiyou Peak	EA-MPU (TrustLite)	Attest-Key
EA-MPU rules		1	1
Register	5528	$278 + (116 \cdot \#r)$	0
Look-Up Table (LUT)	14361	$417 + (182 \cdot \#r)$	0

	Counter	64 bit clock	32 bit clock	SW-clock
EA-MPU rules	1	0	0	0
Register	0	64	32	0
Look-Up Table (LUT)	0	64	32	0

Table 12: Hardware cost per component (#r is the number of protection rules configurable in EA-MPU).

15142 LUTs; see Table 12 columns "Siskiyou Peak" and "EA-MPU". For the 64 bit clock implementation, one additional EA-MPU rule is needed, plus the cost of the clock itself: $116 + 64 = 180$ registers and $182 + 64 = 246$ LUTs, i. e., 2.98% and 1.62% of the overall cost, respectively. For the 32 bit clock with a frequency divider, the cost is $116 + 32 = 148$ registers and $182 + 32 = 214$ LUTs, which means an overhead of 2.45% and 1.41%, respectively.

The SW-clock implementation requires three new EA-MPU rules: $116 \cdot 3 = 348$ registers and $182 \cdot 3 = 546$ LUTs, which is 5.76% and 3.61% of the overall cost, respectively.

6.1.6 *Conclusion*

Prior attestation methods assume a trusted verifier and an untrusted prover. We have shown that verifier impersonation and prover-bound DoS attacks pose serious threats, which have been largely overlooked by prior work. And we formulated a new roaming adversary model, which can defeat classical replay attack mitigation approaches. We developed and evaluated techniques to protect the prover from attacks by this powerful adversary and showed that desired protection can be achieved with low additional hardware cost.

6.2 SEDA: Scalable Embedded Device Attestation

Traditionally, Remote Attestation (RA) is a protocol between two entities, a *verifier* and a *prover*. The latter demonstrates that it is in a known and trustworthy state by sending a status report of its current software configuration. However, this peer-to-peer relation between single prover and single verifier limits RA's scalability and thus its applicability to many new use cases and scenarios, for instance in the Internet of Things (IoT).

Inspired by nature, such large systems of connected and collaborative devices, such as robots, are often referred to as *swarms* [114, 190, 331]. Using traditional, peer-to-peer RA schemes naïvely, i. e., attesting each device of the swarm individually, does not scale to these systems. Furthermore, swarms with dynamic typologies, e. g., robot swarms, or limited connectivity, i. e., swarms where only neighboring devices can directly communicate with each other, impede the adaption of traditional attestation schemes to these systems.

Goals and Contributions. Our goal is the design of an efficient and secure multi-prover RA scheme. It must support dynamic topology of the swarm's devices and ensure that compromised devices cannot evade detection during attestation, i. e., devices of the swarm must be accurately accounted (e. g., honest device must not be counted more than once). Furthermore, the computation and communication cost in large-scale swarms should be minimized while distributing the protocol's workload among all swarm devices.

We present Scalable Embedded Device attestation (SEDA), which to the best of our knowledge, is the first *multi-prover* attestation scheme. It enables efficient attestation of large-scale swarms of connected devices and opened a new line of research inspiring many follow-up works adapting and extending the core idea of SEDA [16, 199, 201, 82].

- We present the first security model for collective attestation of device swarms.
- We design SEDA, the first protocol for multi-prover RA.
- We provide two instances of SEDA for state-of-the-art low-end embedded systems security architectures (SMART [139] and TrustLite [226]). We show how these architectures can be adapted to SEDA with minimal hardware cost.
- In our evaluation we assess the performance of our SEDA instances and show that SEDA scales significantly better than attesting each individual device of a swarm separately.

6.2.1 *System Model and Preliminaries*

This section introduces the assumptions and objectives of SEDA.

6.2.1.1 *System Model and Assumptions*

A *swarm* $\mathcal{S}$ is a set of s heterogeneous embedded devices D_i, i. e., devices with possibly different hardware and software configurations. Each device in $\mathcal{S}$ can be attested via

RA based on a low-end embedded system security architecture [139, 147, 226], i. e., all devices need to satisfy the minimal requirements for secure RA.

Within the swarm devices can communicate with their direct neighbors [114, 190, 331] and the entire swarm is connected, e. g., via a mashed network. Furthermore, $\mathcal{S}$ might be dynamic in terms of membership, i. e., devices can join and leave $\mathcal{S}$, and topology, i. e., device mobility within the swarm leading to changing neighborship relations.

All devices are initialized and deployed by a *swarm operator* $\mathcal{OP}$. $\mathcal{OP}$ is trusted and the initialization and deployment process cannot be manipulated by an adversary.

Swarm Attestation Protocol Objectives and Requirements. *Swarm attestation* should provide a *verifier* $\mathcal{VRF}$ assurance about the swarm's state, i. e., whether the software integrity of all devices, and thus, the integrity of the overall swarm, is given or not. $\mathcal{VRF}$ may be a remote entity and may have no pre-established relations with any device of the swarm. Due to the swarm's dynamics, the topology of $\mathcal{S}$ might be unknown to $\mathcal{VRF}$.

Swarm attestation provides an external verifier $\mathcal{VRF}$ information allowing $\mathcal{VRF}$ to reliably learn whether the swarm is in a correct software state. The correctness of the swarm is given if all devices of the swarm, i. e., all devices deployed by $\mathcal{OP}$, are in a correct software state. The correct software state is defined by $\mathcal{OP}$ and unavailable devices are considered incorrect. In particular, non-responding devices are not operating correctly, and thus, must be assumed to be invalid or compromised.[1]

A secure swarm attestation scheme must fulfill the following requirements:

SA-R1 *Verifiable integrity:* Remote verifiability of the integrity of a swarm $\mathcal{S}$ as a whole.

SA-R2 *Efficiency:* Higher efficiency than attesting each device D_i in $\mathcal{S}$ individually.

SA-R3 *Swarm configuration agnostic:* Configuration details of $\mathcal{S}$ must be oblivious to $\mathcal{VRF}$ (e. g., device types, deployed software versions, and network topology).

SA-R4 *Parallelism:* Support for multiple parallel or overlapping attestation protocol instances.

SA-R5 *Integrity measurements independence:* Independence of the underlying integrity measurement mechanism used to capture the state of devices in $\mathcal{S}$.

The first requirement expresses the core objective of swarm attestation. The second requirement is essential for scalability in large swarms. The third requirement allows swarm attestation to be used in scenarios were no previous relation exists between $\mathcal{S}$ and $\mathcal{VRF}$. Further, requirement SA-R3 is needed in scenarios where the swarm's configuration or details of the swarm's devices should not be disclosed to $\mathcal{VRF}$. For example, in smart factories, the maintenance may be outsourced, and maintenance staff may need to check overall trustworthiness of the production system while the exact setup remains secret [268, 255, 421]. The fourth requirement enables application scenarios where multiple verifiers need to independently verify system integrity without coordination. And the fifth requirement is needed for extensibility and portability, i. e., to

1 Unavailable devices are assumed to be detected by a time-out mechanism at a lower network layer.

support a wide range of single-device attestation mechanisms and to be able to adapt to sophisticated attestation schemes, e. g., those that allow detection of code-reuse attacks.

Device Requirements. To fulfill the first three requirements listed above, the swarm's devices D must provide mechanisms to be remotely verified, to verify other D_j, and to aggregate results securely. This implies that all D_i have to fulfill the requirements listed below. This includes the requirements imposed by single device attestation solutions [139, 147].

D-R1 *Integrity measurement:* D_i must provide a mechanism to measure and report its software state in an authentic way.

D-R2 *Integrity reporting:* D_i must provide means to send an aggregated integrity measurement report to $\mathcal{VRF}$.

D-R3 *Secure storage and processing:* D_i must be able store and process data, for instance cryptographic secrets, as part of the swarm attestation protocol.

Subsequently we provide the design and prototype implementation of our swarm attestation protocol SEDA. We show that SEDA fulfills all requirements SA-R1 to SA-R5, and show how to utilize two embedded security architectures – SMART [139] and TrustLite [226] – to implement SEDA and fulfill requirements D-R1 to D-R3.

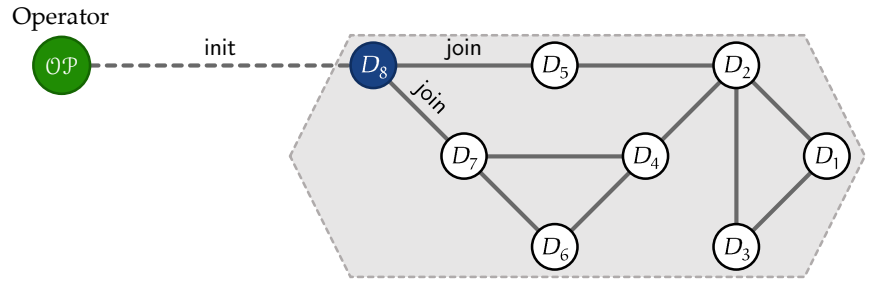
6.2.1.2 *Adversary Model*

Our adversary model for SEDA is based on the same assumptions common in the attestation literature [217, 342, 346, 139]. In particular, the adversary $\mathcal{ADV}$ can compromise and manipulate the software of any device D_i, i. e., prover, in $\mathcal{S}$. Further, $\mathcal{ADV}$ can eavesdrop on, manipulate, and inject any message between devices, as well as any message between $\mathcal{VRF}$ and any device D_i.

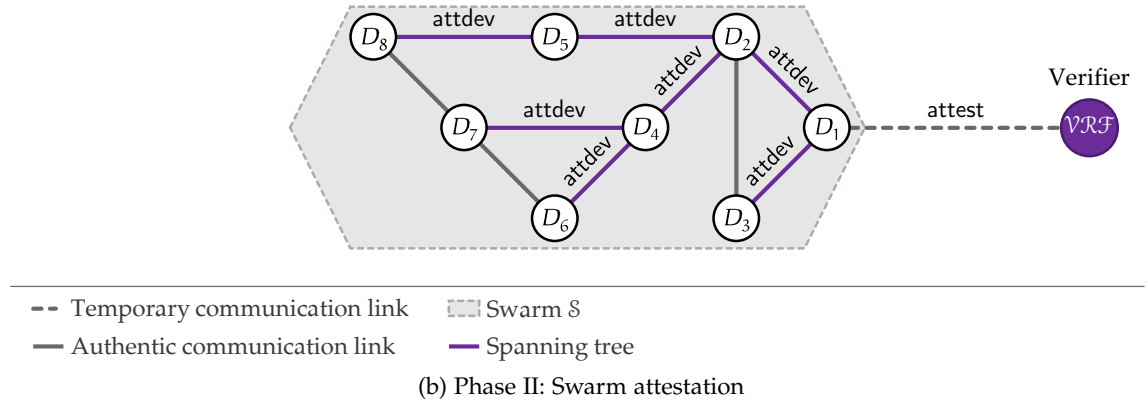
The adversary $\mathcal{ADV}$ cannot physically tamper with any device D_i and the Trusted Computing Base (TCB) of any devices is ever compromised by $\mathcal{ADV}$. We consider Denial-of-Service (DoS) attacks out of scope. $\mathcal{ADV}$ typically aims to remain stealthy – an adversary that does not mind being detected has no incentive to overcome the attestation scheme in the first place. Nevertheless, we discuss different approaches to mitigate or detect both, physical attacks as well as DoS attacks in Section 6.2.7 and Section 6.2.5, respectively.

6.2.2 *SEDA Protocol*

Our swarm attestation protocol SEDA works in two phases. In the first phase, the swarm is prepared (or modified), e. g., by introducing devices into the swarm. The first phase is performed off-line, it is executed once for each device D_i that gets integrated into the swarm, i. e., each device is *initialized* and *registered*. For a swarm $\mathcal{S}$ where the devices do not change over time, this initial phase is executed only once before $\mathcal{S}$ gets operational. In the second, on-line phase, the actual attestation of the swarm is performed. This phase is



(a) Phase I: Device initialization and registration



(b) Phase II: Swarm attestation

Figure 33: SEDA swarm attestation in 8-device swarm.

executed repeatedly, i. e., whenever an attestation request from a verifier $\mathcal{VRF}$ arrives, all device D_i of $\mathcal{S}$ are attested. The attestation reports of the individual D_i are accumulated and the resulting swarm attestation report is sent to $\mathcal{VRF}$.

Figure 33a shows phase I of the SEDA protocol. A new device D_8 is integrated into the existing swarm $\mathcal{S}$ consists of seven devices $D_1, \ldots, D_7$. The operator $\mathcal{OP}$ initializes the new device D_8. Then, D_8 executes the *join* protocol with its neighboring devices D_5 and D_7.

Phase II is shown in Figure 33b. $\mathcal{VRF}$ initiates an attestation of $\mathcal{S}$ by sending an attestation request to D_1. The swarm's devices execute the *attdev* protocol with their neighboring devices such that each device $D_1, \ldots, D_8$ is attested exactly once, by constructing a spanning tree. Within the spanning tree parents attest their children. The accumulated results are collected by D_1 and reported to $\mathcal{VRF}$.

6.2.2.1 *Phase I: Device Initialization and Registration*

In SEDA's off-line phase devices are first initialized. Afterwards, devices in $\mathcal{S}$ register with their neighboring devices using the join protocol.

Device Initialization. Each device D_i in a swarm $\mathcal{S}$ is initialized by the operator $\mathcal{OP}$ in order to be prepared to participate in the swarm attestation protocol (see below Section 6.2.2.2). Therefore, $\mathcal{OP}$ installs a software configuration c_i (represented by a hash

digest of dev_i's software binaries) and a code certificate $cert(c_i)$ on D_i. $cert(c_i)$ is signed by $\mathcal{OP}$ guaranteeing that c_i is a valid software configuration of D_i. Furthermore, D_i is initialized with a signing key pair (sk_i, pk_i).[2] An identity certificate $cert(pk_i)$, also signed by $\mathcal{OP}$, binds pk_i to D_i and authenticates pk_i, i. e., guarantees that pk_i belongs to D_i.

To be able to validate the certificates of other devices in $\mathcal{S}$, each D_i is initialized with the operator's public key $pk_{\mathcal{OP}}$.

Finally, the list of active session identifiers $\mathcal{Q}_i$ of a new device D_i is initialized as an empty list.

Device Registration. When a device D_i initially joins a swarm $\mathcal{S}$ or changes its positions within $\mathcal{S}$, i. e., when D_i becomes new neighboring devices, it executes the join protocol with each new neighbor device D_j. In Figure 33a D_8 joins swarm $\mathcal{S}$ and runs the registration protocol with its neighbors D_5 and D_7. Through the join protocol, D_i learns each of its neighbors' expected configuration c_j by receiving D_j's code certificate $cert(c_j)$. D_i verifies the received certificates using $pk_{\mathcal{OP}}$ and if verification succeeds, D_i stores c_j for later validation of D_j's attestation reports. If verification of $cert(c_j)$ fails, D_i does not accept D_j as a neighbor.[3]

Additionally, D_i establishes a shared key, called attestation key k_{ij}, with each if its neighbors D_j as part of the join protocol, resulting in a set of attestation keys $\mathcal{K}_i$ containing a shared key per neighbor device D_j. Key establishment can be done using standard authenticated key agreement protocol based on devices' sk_i, sk_j, $cert(pk_i)$, and $cert(pk_j)$. Alternatively, shared keys between neighbor devices can be established using techniques such as key pre-distribution [80]. SEDA can be used with any key establishment protocol that provides integrity of key agreement and secrecy of generated keys.

6.2.2.2 *Phase II: Swarm Attestation*

SEDA's online phase involves three protocols: attest, attdev and clear. attest executes between the verifier $\mathcal{VRF}$ and an arbitrary device $D_1 \in \mathcal{S}$, called *initiator*. The attest protocol initiates the attestation of $\mathcal{S}$ and is initiated by $\mathcal{VRF}$.

The attdev protocol is executed between neighboring devices in $\mathcal{S}$. Starting from the initiator D_i, all devices of $\mathcal{S}$ are attested recursively using attdev. All attestation results are accumulated and eventually received by D_1, which reports the result to $\mathcal{VRF}$ as part of attest.

The clear protocol is executed at the end of a swarm attestation session to remove all temporary data from all $D_i \in \mathcal{S}$

Swarm Attestation. $\mathcal{VRF}$ starts attestation of $\mathcal{S}$ by sending an attestation request attest (containing a random challenge) to the initiator device D_1. $\mathcal{VRF}$ can randomly select any device in $\mathcal{S}$ as initiator D_1. $\mathcal{VRF}$ might also chose D_1 depending its preference, e. g., the closed $D_i \in \mathcal{S}$ based its location. $\mathcal{VRF}$ can be either remote (e. g., connected via Internet) or within direct communication range of one or more swarm devices.

2 Device certificates are issued and managed by $\mathcal{OP}$ using its own Public Key Infrastructure (PKI).

3 If D_i's software is updated after it joins $\mathcal{S}$, D_i must communicate its new code certificate to all its neighbors, otherwise the new configuration will not be accepted in a subsequent attestation.

attest operates as follows: The verifier $\mathcal{VRF}$ starts the protocol by sending an attestation request including a nonce N to D_1. N prevents replay attacks on the communication between $\mathcal{VRF}$ and D_1. When receiving an attestation request D_1 creates a new attestation session by generating a new global session identifier q. D_1 then executes attdev with all its neighbors, including q. All neighbor devices recursively execute attdev with their neighbors. q is included in all subsequent executions of attdev to identify the current swarm attestation protocol instance and to build a spanning tree to prevent circular attestation among the swarm's devices. When a device D_j receives the attestation results for all neighbors, it initiated attdev with, it accumulates the results and reports back to the device that initiated attdev with D_j. This way, D_1 eventually receives the accumulated attestation reports of all other devices in $\mathcal{S}$. D_1 accumulates all received reports into (β, τ), with β being the number of successfully attested devices and τ being the total number of attested devices. It computes a signature σ over (β, τ) and its own software configuration c_1'. As the result of the swarm attestation, D_1 sends its device identity certificate $\text{cert}(pk_1)$, code certificate $\text{cert}(c_1)$, (β, τ), c_1', and σ to $\mathcal{VRF}$. Using $\mathcal{OP}$'s public key $pk_{\mathcal{OP}}$, $\text{cert}(pk_1)$ and $\text{cert}(c_1)$, $\mathcal{VRF}$ authenticates pk_1 and c_1, and verifies σ. Attestation succeeds if σ is correct and (β, τ) show that all $D_i \in \mathcal{S}$ are in a software state matching their corresponding code certificates $\text{cert}(c_i)$.

If $\mathcal{VRF}$ does not receive a response from D_1 (e. g., due to a connection time-out) swarm attestation fails.

After receiving responses from all its neighbors and responding to $\mathcal{VRF}$, D_1 starts the clear protocol to finish the current protocol session by delete q from all devices in $\mathcal{S}$.

Single Device Attestation. SEDA uses the attdev protocol to perform attestation between the individual devices of $\mathcal{S}$. As described above, each swarm attestation instance has a global session identifier q that is generated by the initiator device D_1. q is used to construct a *spanning tree* over all devices of the swarm. Each device D_j maintains a list of active sessions by storing each q it receives in a list $\mathcal{Q}_j$. Furthermore, by distributing q in $\mathcal{S}$ a spanning tree T is created: when D_j receives a new q from another device D_i it accepts D_i as its parent in T. D_j sends q to all its remaining neighbors D_i ($D_i \neq D_j$) to expand T further, eventually covering all $D \in \mathcal{S}$. However, the construction of T can be adjusted, e. g., by setting a maximum number of children each device may have, limiting the fan out of T and leading to a growth in height. This is a way to optimize SEDA's performance (cf. Section 6.2.4). It allows the transformation of different network topologies into balanced spanning trees. In particular, the spanning tree T is constructed from the communication graph of $\mathcal{S}$, i. e., two devices can be neighbors in T if they are neighbors in the communication graph.

Within T each device attests all its children, i. e., a parent device D_j sends an attestation request to each D_k that is its child in T. The attestation request contains a nonce N_{jk} and the session identifier q. Once a device D_j receives the attestation reports (including the attestation information accumulated by each child) from all its children this information is accumulated, resulting in a report for the subtree T_j with root D_j. D_j then sends the accumulated attestation for T_j along with the attestation report for itself to its parent D_i.

To authenticate the attestation report from a child D_k and ensure the report's integrity, D_j uses the attestation key k_{jk}, which was established in the join protocol. To validate

D_k's attestation report D_j uses the reference software configuration c_k, which D_j learned during the join protocol.

The result of the attdev protocol between D_i and its child D_j is, (i) whether attestation of D_j was successful, i. e., D_j's current software configuration matches its reference software configuration c_j, (ii) the number of device τ_j in the subtree T_j rooted at D_j (excluding D_j), and (iii) the number of devices β_j in T_j that have been successfully attested (again, excluding D_j). D_i accumulates the results it received from all its children $D_{j_1}, \ldots, D_{j_n}$. τ_i is the sum of $\tau_{j_1}, \ldots, \tau_{j_n}$ plus the number of D_i's children n: $\tau_i = \tau_{j_1} + \cdots + \tau_{j_n} + n$. β_i is the number of successfully attested devices reported by D_i's children $\beta_{j_1}, \ldots, \beta_{j_n}$ plus the number m of successfully attested children $D_{j_1}, \ldots, D_{j_n}$: $\beta_i = \beta_{j_1} + \cdots + \beta_{j_n} + m$.

If q is already in the list Q_j of active session identifiers, D_j responds with $\beta_j \leftarrow \perp$ and $\tau_j \leftarrow \perp$. When D_j does not response to D_i in an attdev protocol instance D_i considers the attestation of D_j failed, and D_i sets $\beta = 0$ and $\tau = -1$ to prevent double-counting.

Figure 33b shows a sample swarm with eight devices: $D_1, \ldots, D_8$. The spanning tree is denoted by violet lines between devices. The root is D_1, which is selected by $\mathcal{VRF}$ to be the initiator.

Clear. When an attestation session is finished all corresponding temporary data are removed from all device in $\mathcal{S}$. Therefore, the clear protocol is executed between neighboring devices, i. e., D_i sends the q of the finished session to D_j, authenticated with k_{ij}. When received, D_j removes q from its list Q_j of active session identifiers and runs clear protocol with each child in the previously constructed spanning tree.

6.2.3 *Prototype and Implementation*

In this section we discuss our implementations of SEDA based on two different security architectures for low-end embedded systems, SMART [139] and TrustLite [226]. These security architectures have different security properties and provide different functional capabilities, while relying on minimal hardware assumptions.

SMART-based Implementation. The SMART architecture provides the minimal set of functionalities to enable RA for low-end embedded devices [139, 147].

SMART requires two main components that enable secure RA: (1) Read-Only Memory (ROM) to protect the integrity of the attestation routine and the attestation key k,[4] and (2) a Memory Protection Unit (MPU) that controls access to k, stored in ROM, to ensure confidentiality for k, i. e., only the trusted and immutable attestation routine can access k.

Our first SEDA implementation extends the functionality of SMART. The ROM contains all protocol routines of SEDA, i. e., join, attdev, attest and clear. Furthermore, the device's singing key sk is stored in ROM. Similar to the original SMART design, access to the device key (sk) is controlled by the MPU, it is accessible only to SEDA's routines.

The MPU of SMART has been extended to enforce three additional access rules. During join protocol shared keys with all neighboring devices are established and stored in $\mathcal{K}$. Both, the integrity and confidentiality of $\mathcal{K}$ is guaranteed via access control enforced

4 One-time programmable ROM allows initializing each device with a distinct device-specific key during device initialization.

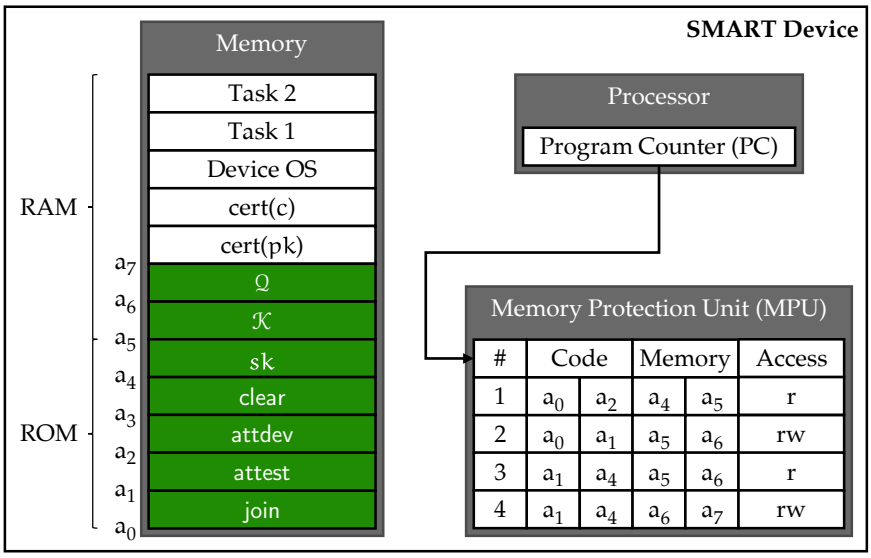


Figure 34: SEDA implementation based on SMART [139]

by the MPU. Similarly, the list of active session identifiers $\mathcal{Q}$ must be protected by an additional MPU rule.

Figure 34 shows the memory layout and the MPU access rules for our SMART-based SEDA implementation. In particular, the first MPU rule ensures that sk is only readable by join and attest, the only protocols in SEDA using sk. The second MPU rule allows the join protocol to read and write $\mathcal{K}$, i. e., join can add shared keys with neighboring devices to $\mathcal{K}$. Rule number three grants attest, attdev and clear the right to read the shared attestation keys from $\mathcal{K}$. Finally, the fourth rule enables attest, attdev and clear to manage the set of active sessions $\mathcal{Q}$. All accesses to the protected memory regions (sk, $\mathcal{K}$, and $\mathcal{Q}$) are denied.

All other SEDA related data that need to be stored on a device are public information (cert(c) and cert(pk)) and, thus, do not need to be protected.

TrustLite-based Implementation. TrustLite [226] is a security architecture for embedded systems, based on Intel's Siskiyou Peak research platform [324]. It enables execution of arbitrary code, (e. g., attest and attdev) isolated from the rest of the system. Such isolated code chunks are called *trustlets*. Similar to SMART, an extended MPU ensures that data can be accessed only by code of trustlets that own that data. Data access permissions depend on the currently executing code – therefore TrustLite's MPU is called Execution-Aware Memory Protection Unit (EA-MPU). Additionally, the EA-MPU can be used to control access to hardware components, such as peripherals (cf. Section 2.1.1). Authenticity and confidentiality of both code and data of trustlets are ensured via secure boot.

TrustLite can be seen as a generalization of SMART. The main difference is that in TrustLite the EA-MPU's memory access control rules can be configured dynamically, as required by trustlets. In contrast, memory access control rules of SMART's MPU are

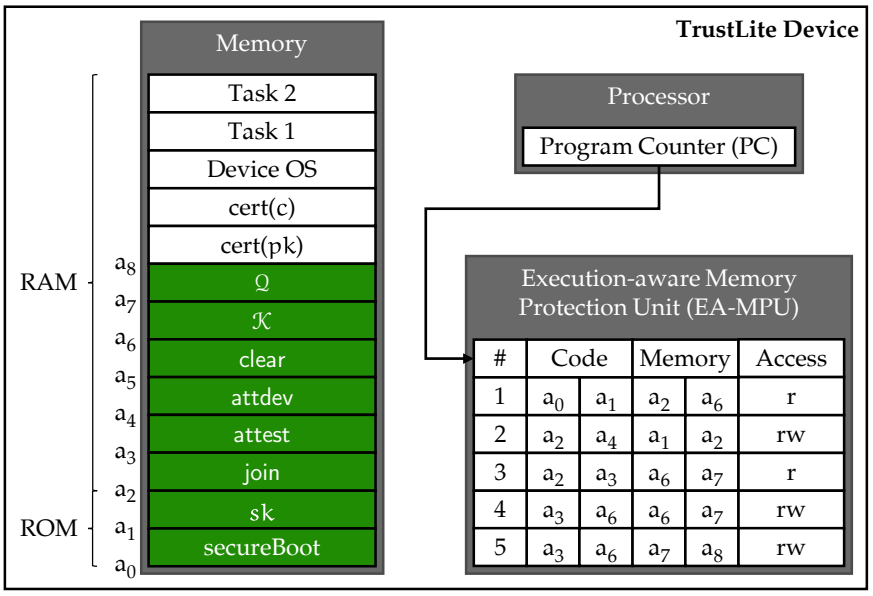


Figure 35: SEDA implementation based on TrustLite [226]

static. Also, TrustLite supports interrupt handling for trustlets, while security-critical code in ROM of SMART must not be interrupted during execution.

We implemented SEDA on TrustLite as trustlets, i. e., join, attest, attdev, and clear are each implemented as individual trustlets. Integrity of these trustlets is ensured by the secure boot component secureBoot.

The EA-MPU controls access to ROM and Random Access Memory (RAM) such that only the SEDA trustlets can access secret data. Figure 35 shows the memory layout and the EA-MPU access rules for our TrustLite-based SEDA implementation. Since the SEDA protocols are not stored in ROM, their integrity must be ensured via secure boot. In particular, the first rule enables the secure boot code to read and verify the integrity of the SEDA protocol implementations join, attest, attdev, and clear. The second rule grants join and attest read access to the device key sk. The set of attestation keys $\mathcal{K}$ can only be written by join (rule 3) and read by attest, attdev and clear (rule 4). Finally, the fifth rule enables attest, attdev and clear to manage the set of active sessions $\mathcal{Q}$.

6.2.4 *Performance Evaluation*

We evaluated computation, memory, communication, and energy costs of SEDA based on our two implementations described in Section 6.2.3.

For the evaluation, we assume that the swarm is static throughout protocol execution. However, in Section 6.2.6 we sketch out a protocol extension to handle *highly dynamic* swarms. Our evaluation results do not include the join protocol, as it is not part of the actual attestation process and is executed rarely, perhaps only once per device. However, the performance of join can be estimated based on the measurement results for the attest

protocol. For both protocols the asymmetric cryptography operations contribute the major part of the computational cost.

Computation Cost. The dominating factor of SEDA's computational cost is the use of cryptography operations, such as Keyed-Hash Message Authentication Codes (HMACs), for the communication as well as for measuring a device's configuration.

The initiator device D_1 communicates directly with the verifier $\mathcal{VRF}$. This communication is secured via a digital signature, which D_1 has to compute. Additionally, D_1 needs to verify the messages it receives from its children in SEDA's spanning tree T, with h_1 being the number D_1's children in T. Per child, two messages have to be verified, hence $2 \cdot h_1$ HMACs have to be calculated by D_1. Finally, D_1 needs to measure its own software configuration by calculating an HMAC over relevant memory locations.

All other devices of the swarm ($D_j \in \mathcal{S}, D_j \neq D_1$) need to verify the messages of their children. D_j receives and verifies messages from its h_j children in spanning tree T, two messages each, hence D_j has to calculate $2 \cdot h_j$ HMACs. Additionally, D_j has to calculate two more HMACs for the messages it sends to it parent.

Communication Cost. Our implementation of SEDA uses HMACs based on Secure Hash Algorithm 1 (SHA1) and ECDSA signature scheme with $\ell_{sign} = 320$, i. e., 20 Byte and 40 Byte respectively. Nonces are 20 Byte, the values for β and τ are stored in 64 bit counters, i. e., 8 Byte each, and certificates are 20 Byte + 40 Byte = 60 Byte. Therefore, D_1 has to send 176 Byte + $h_1 \cdot 48$ Byte and receive 20 Byte + $h_1 \cdot 56$ Byte, with h_1 being the number D_1's of children in SEDA's spanning tree T. All other devices of the swarm ($D_j \in \mathcal{S}, D_j \neq D_1$) have to send 68 Byte + $h_j \cdot 56$ Byte and receive 56 Byte + $h_j \cdot 68$ Byte (h_j is the number of D_j's children in T).

Memory Cost. Each swarm device $D_i \in \mathcal{S}$ has to store (1) the global identifier q of all concurrent swarm attestation protocol instances, (2) its device key, i. e., signing key pair (sk_i, pk_i), (3) its identity certificate $(cert(pk_i))$, (4) the code certificate for is current software configuration $cert(c_i)$, and (5) the set of attestation keys $\mathcal{K}$ is has established with its neighbors via the join protocol.

Each q is 8 Byte, i. e., for s sessions D_i needs to store $s \cdot 8$ Byte. The device key is 80 Byte (320 bit for each sk_i and pk_i). Both, identity certificate and configuration certificate are each 60 Byte. Each key in $\mathcal{K}$ is 20 Byte, i. e., with g_i neighboring devices D_i needs to store $g_i \cdot 20$ Byte.

Hence, in total, D_i has to store $s \cdot 8$ Byte + 200 Byte + $g_i \cdot 20$ Byte, for s concurrent sessions and g_i neighboring devices.

Typical low-end embedded devices such as the TI MSP430, as targeted by SEDA, provide non-volatile Flash memory in the order of 1 MB or more. For scenarios where each device has twelve or less neighbors, 512 kB of non-volatile memory is already sufficient to store all data needed

Run-time. To increase efficiency, SEDA is designed to balance parallelism and dependencies among the swarm's devices. Our SEDA design allows all devices at the same height in the spanning tree T to operate in parallel. However, the root D_i of each subtree needs to collect, verify and accumulate the results of all its child devices D_j, which leads to a delay linear in the number of child devices D_j. In particular, devices at height l can only

proceed after receiving input from all their child devices at height $l-1$. Thus, the overall run time of SEDA is linear in the height of the spanning tree T, and therefore logarithmic in the size of $\mathcal{S}$.

The worst-case run time t for SEDA, given each device that is not a leave in T has sufficient neighboring devices to have c children, is when all child devices D_j answer at the same time and D_i can therefore only start its processing after receiving all answers:

$$t \leqslant 196 \cdot t_{tr} + t_{sign} + d \cdot t_{prng} + d \cdot 84 \cdot t_{tr} + d \cdot 4 \cdot t_{mac}$$

with t_{tr} being the time required to transmit one byte of data, t_{sign} time needed to compute a signature sign, t_{prng} time to generate 20 random bytes using a Pseudorandom Number Generator (PRNG) function, and t_{mac} time required to compute a Message Authentication Code (MAC) or verify a MAC, i. e., for mac and vermac respectively.

Our simulation-based evaluation shows that SEDA performs very well even for large swarms with up to 100 000 devices. A detailed evaluation for different network topologies is provided in our conference publication [34], in which we also provide an evaluation of SEDA's energy cost for each swarm device.

6.2.5 *Security Analysis*

A swarm attestation scheme is secure if a verifier $\mathcal{VRF}$ accepts an attestation report as valid *only if all* device $D_1, \ldots, D_n \in \mathcal{S}$ are in a software configuration admissible by the swarm operator $\mathcal{OP}$. Admissible software configurations are authenticated by $\mathcal{OP}$ via configuration certificated cert(c), which are signed by $\mathcal{OP}$.

The adversary $\mathcal{ADV}$ can modify the software of arbitrary devices of the swarm ($D_{\mathcal{ADV}} \subset \mathcal{S}$), i. e., compromise D_a. Furthermore, $\mathcal{ADV}$ can eavesdrop on, delete, and modify any message sent between devices in the swarm as well as messages sent between $D_a \in D_{\mathcal{ADV}}$ ($a = 1, \ldots, m; m = |D_{\mathcal{ADV}}|$, with $|D_{\mathcal{ADV}}|$ number of adversary-controlled devices) and $\mathcal{VRF}$.

The adversary $\mathcal{ADV}$ succeeds when it can manipulate at least one device while the verifier $\mathcal{VRF}$ accepts the resulting attestation report as valid.

The adversary can attack the system in four different ways: (1) Compromise the software of the initiator device D_1, which interacts with $\mathcal{VRF}$. (2) Compromise the software of any other device of the swarm $D_j \in \mathcal{S}, D_j \neq D_1$. (3) Manipulate communication between D_1 and $\mathcal{VRF}$. Or (4) manipulate the communication between the swarm's devices.

These four cases, and the combination thereof represent all possibilities for $\mathcal{ADV}$ to tamper with the swarm attestation. Preventing communication or a device's participation in SEDA's protocols does not facilitate $\mathcal{ADV}$ as these actions will be detected by SEDA.

Case 1: $\mathcal{ADV}$ compromises the initiator device D_1, i. e., modifies the software configuration of D_1 to a different software configuration c_1'. According to SEDA's adversary model (Section 6.2.1.2), the code performing integrity measurements on D_1 and the code of attest belong to D_1's TCB, hence, $\mathcal{ADV}$ cannot tamper with them. Therefore, D_1's measured configuration c_1' will differ from the reference software configuration c_1 in cert(c_1). The measured configuration c_1' is signed by D_1 with its device key sk,

which is only available to the device's TCB, i. e., inaccessible for $\mathcal{ADV}$. Thus, $\mathcal{ADV}$ has three options: (a) $\mathcal{ADV}$ has to change the configuration measurement in the attestation report to a valid measurement, i. e., manipulate the communication between D_i and $\mathcal{VRF}$. (b) $\mathcal{ADV}$ has to generate a forged configuration certificate $\text{cert}(c_1')$ without knowledge of $\mathcal{OP}$'s secret signing key $sk_{\mathcal{OP}}$. Or (c) $\mathcal{ADV}$ has to find a software configuration that operates as intended by $\mathcal{ADV}$ and leads to the same measurement as the correct software configuration, i. e., find a second pre-image for the hash function used to measure the devices' software configurations. Therefore, $\mathcal{ADV}$ can only succeed if the used signature scheme or the used measurement function is insecure, or if $\mathcal{ADV}$ can manipulate the communication between D_1 and $\mathcal{VRF}$, discussed below (Case 3).

Case 2: $\mathcal{ADV}$ compromises, without loss of generality, one device $D_j \in \mathcal{S}, D_j \neq D_1$, i. e., modifies the software configuration of D_j to a different software configuration c_j'. Assuming $D_i \in \mathcal{S}$ is D_j's parent in the spanning tree that is built during swarm attestation, i. e., D_j sends its attestation report to D_i and D_i verifies the report. According to SEDA's adversary model (Section 6.2.1.2), the code performing integrity measurements on D_j and the code of attest belong to D_j's TCB, hence, $\mathcal{ADV}$ cannot tamper with them. Therefore, D_j's measured configuration c_j' will differ from the reference software configuration c_j in $\text{cert}(c_j)$. To prevent detection, $\mathcal{ADV}$ has three options: (a) $\mathcal{ADV}$ has to change the configuration measurement in the attestation report to a valid measurement, i. e., manipulate the communicated data between D_i and D_j. (b) $\mathcal{ADV}$ has to generate a forged configuration certificate $\text{cert}(c_j')$ without knowledge of $\mathcal{OP}$'s secret signing key $sk_{\mathcal{OP}}$. Or (c) $\mathcal{ADV}$ has to find a software configuration that operates as intended by $\mathcal{ADV}$ and leads to the same measurement as the correct software configuration, i. e., find a second pre-image for the hash function used to measure devices' software configurations. This means, $\mathcal{ADV}$ can only succeed if the used signature scheme or the used measurement function are insecure, or when $\mathcal{ADV}$ can manipulate the communication between the swarm's devices, discussed below (Case 4).

Case 3: $\mathcal{ADV}$ manipulates communication between D_1's and $\mathcal{VRF}$. All information of the swarm attestation protocol is authenticated and integrity protected, using secrets only available to D_1's TCB, in particular, the attestation report is signed with D_1's device key sk_1. Therefore, $\mathcal{VRF}$ can validate the authenticity and integrity of all information received preventing $\mathcal{ADV}$ from modifying or inserting any messages that will be accepted by $\mathcal{VRF}$ if the used signature scheme is secure. $\mathcal{ADV}$ cannot inject old messages with a valid signature as each swarm attestation session is started with new nonce N allowing $\mathcal{VRF}$ to detect replayed messages. To validate messages, $\mathcal{VRF}$ needs knowledge of D_1's public key pk_1, which is authenticated by $\mathcal{OP}$ through the provided certificate $\text{cert}(pk_1)$. Thus, $\mathcal{ADV}$ can only convince $\mathcal{VRF}$ to accept a wrong, adversary chosen public key $pk_{\mathcal{ADV}}$ if the used signature scheme is insecure and $\mathcal{ADV}$ can create a valid certificate $\text{cert}(pk_{\mathcal{ADV}})$ without knowledge of $\mathcal{OP}$'s secret signing key $sk_{\mathcal{OP}}$.

Case 4: $\mathcal{ADV}$ manipulates the communication between devices of the swarm $\mathcal{S}$. Without loss of generality, we assume $\mathcal{ADV}$'s goal is to manipulate the communication between D_i and D_j, where D_i sends a message to D_j. All information of the swarm attestation protocol are authenticated and integrity protected, using secrets only available to D_i's

and D_j's TCB, using Keyed-Hash Message Authentication Code (HMAC) with the shared attestation key k_{ij}. Hence, D_j can validate the authenticity and integrity of all information received from D_i preventing $\mathcal{ADV}$ from modifying or inserting any messages that will be accepted by D_i if the used HMAC scheme is secure. $\mathcal{ADV}$ cannot inject old messages with a valid HMAC as each interaction request belonging to a new swarm attestation session is started with new nonce N allowing D_j to detect replayed messages.

The shared attestation key k_{ij} is established between D_i and D_j as part of the join protocol. $\mathcal{ADV}$ can get knowledge to the shared key or convince D_i or D_j to accept another key, known to the $\mathcal{ADV}$, only if the used key establishment protocol is insecure.

6.2.6 *Protocol Extensions*

In this section we discuss possible variants and extensions of SEDA. In particular, we discuss options to strengths SEDA against attacks that go beyond the scope of the adversary model (cf. Section 6.2.1.2), which we considered so far in this work.

Identifying Compromised Devices. In many applications it may be beneficial, and in some it might be even necessary, to identify devices that have been compromised. SEDA can be easily extended to report the identifiers of devices that were detected to have an invalid software configuration, i. e., their software integrity could not be verified. Whenever a device D_i detects that one of its child devices D_j reported a software configuration c_j' that does not match the certified software configuration $\text{cert}(c_j)$, D_i includes the identifier of D_j to its report. D_i's parent will pass the identifier on and include the identifiers of other compromised devices, until eventually, $\mathcal{VRF}$ receives a complete list of identifiers of all devices that could not be attested successfully. However, this approach increases message complexity of the reports, as the message size will depend on the number of compromised devices found. Hence, this approach is best suited for applications where the number of compromised devices is expected to be low.

Devices with Different Priorities. In different applications the criticality, e. g., relevance for the correct overall operation of the system, of some devices might be higher than that of others. For example, in Wireless Sensor Networks (WSNs) the cluster head is crucial for the operation of the overall system. Hence, its correctness, i. e., software integrity, is more important than that of individual sensor nodes. SEDA can account for the different criticality of individual swarm devices by assigning weights to the attestation results of nodes. Attestations of high-priority devices are factored in the overall result by incrementing the counters β and τ with a weighted factor.

Random Sampling. SEDA's performance can be improved by attesting only a randomly sampled statistically representative subset $\mathcal{S}' \subset \mathcal{S}$ rather than attesting all devices of a swarm $\mathcal{S}$. This approach is particularly suitable for very large swarms where already a relatively small subset $\mathcal{S}'$ is sufficient to detect compromised devices with high probability, i. e., that the verifier $\mathcal{VRF}$ gets assurance that all devices in the swarm are running authentic software with high probability.

Random sampling can be integrated in SEDA as follows: $\mathcal{VRF}$ sends in the attest protocol, along with its attestation request and nonce N, the desired sample set size

z. In the attdev protocol, all devices will – during the construction of the spanning tree – broadcast z along with the global session identifier q. Using a global deterministic function, which takes q, z and the size of the swarm $s = |\mathcal{S}|$ as inputs, each device learns whether it is part of $\mathcal{S}'$. Also, the parent device D_i of each device D_j learns whether $D_j \in \mathcal{S}'$, i. e., if D_j needs to provide an attestation report. If a $D_j \in \mathcal{S}'$ does not provide an attestation report to its parent D_i, D_j is accounted as a failed attestation. Finally, all attestation results of devices in $\mathcal{S}'$ are accumulated. $\mathcal{VRF}$ receives an accumulated report containing the total number of attested devices in $\mathcal{S}'$ (β) and the number of successfully attested device (τ).

As a result, $\mathcal{VRF}$ is assured that – with a certain confidence level and confidence interval depending on the size of $\mathcal{S}'$ – the attestation result of $\mathcal{S}'$ reflects the state of $\mathcal{S}$. For instance, attesting only approximately 9 % of all devices in swarms with more than 100 000 devices is sufficient to achieve a confidence level of 95 % and a confidence interval of 1 %.

Software Updates. SEDA can be used to verify whether a software update has been correctly deployed. Specifically, when new software is installed on a device D_i, leading to a new software configuration of the device c_i^*, also a new configuration certificate $\text{cert}(c_i^*)$ is installed on the device. Afterwards, D_i sends the $\text{cert}(c_i^*)$ authenticated with the keys in $\mathcal{K}_i$ to all its neighbors. The neighboring devices store the updated software configuration c_i^*, after successfully verifying the certificate $\text{cert}(c_i^*)$. If the verification fails they retain the configuration c_i, which they had stored before. To prove the successful update of D_i's software, D_i can attest itself to all its neighbor devices D_j using their shared attestation keys k_{ij}, similar to the attestation performed as part of the attdev protocol. D_i can also attest itself to an external verifier $\mathcal{VRF}$, using its secret key sk_i, similar to the attestation of the initiator device in the attest protocol.

The adversary $\mathcal{ADV}$ might want to install an older software version, which might contain exploitable security vulnerabilities, using a roll-back attack. In a roll-back attack the adversary $\mathcal{ADV}$ installs an older software version for which $\mathcal{ADV}$ possesses a still valid configuration certificate. However, outdated software version, due to a roll-back attack can be detected by $\mathcal{VRF}$ when attesting the swarm.

Highly Dynamic Swarms. SEDA can be used with highly dynamic swarms, i. e., swarms where the topology is changing frequently, *even while* the swarm attestation protocol is executing. For this, SEDA needs to create a spanning tree across *virtual* neighbors. After a parent-child relation between two devices has been established the topology changes and the two devices lose their direct connection to each other. Using an appropriate routing mechanism ensures that messages between child and parent devices are still delivered, allowing the SEDA protocol to continue. However, this approach naturally increases SEDA's communication overhead since messages must be sent over multiple hops.

Denial-of-Service (DoS) Attack Mitigation. DoS attacks are hard to mitigate in general. The design of SEDA uses preferably symmetric cryptography, which is less resource consuming and therefore less appealing for DoS attacks. However, SEDA's sub-protocols join and attest require asymmetric cryptography, which could be misused by $\mathcal{ADV}$ to launch a DoS attack. For instance, a compromised device D_i could repeatedly invoke the

join protocol on a neighboring device D_j, forcing D_j to validate the identity certificate D_i sends, even if D_i sends an invalid certificate. This would waste D_j's resources, since verifying public key certificates is computational expensive, in particular for small embedded devices as shown in our evaluation (cf. Section 6.2.4).

Such DoS attacks can be mitigated either by limiting the rate at which join requests are accepted and processed, or processing join request with low priority. With our real-time capable security architecture for embedded systems TyTAN (cf Section 3.1), certain events (e. g., join requests) can be handled with low priority. This allows system resources to be preferentially allocated to more critical tasks, while assuring that only otherwise idle Central Processing Unit (CPU) cycles are dedicated to processing (potentially malicious) join requests.

6.2.7 *Physical Attacks*

There are many settings and use cases for swarms in which physical attacks are either impossible or unlikely, and therefore it is reasonable to have an adversary model that excludes physical attacks (cf. Section 6.2.1.2). Examples for such scenarios include avionics, maritime traffic, or satellites, where the devices are typically out of reach for the adversary $\mathcal{ADV}$ or have a secure perimeter, e. g., security measures as found in airports. In other scenarios, however, it might be unrealistic to assure physical security for *all* devices of a swarm. For instance, devices operating in public areas or even in hostile environments, e. g., military drones and robots or traffic infrastructure such as traffic lights, vehicles etc.

With advanced physical attacks, an adversary $\mathcal{ADV}$ might be able to compromise a device's TCB and extract all its secrets, such as the device's signing key. In SEDA, the aggregation of attestation results is performed by the individual devices of the swarm, hence, a device with a compromised TCB can evade its own detection, and additionally, it can manipulate the attestation information about other devices, in particular, it can forge all information regarding any of its descendants in SEDA's spanning tree. Furthermore, using secret information, in particular keys, extracted from a compromised device, $\mathcal{ADV}$ can create clones of the compromised device placing them across the swarm, i. e., allowing $\mathcal{ADV}$ to effectively control many (fake) devices in the swarm.

To address these risks, we sketch out different mitigation techniques, considering an extended adversary model, which assumes an adversary that can capture and physically attack device to extract keys or modify a device's TCB.

PUF-based Attestation. Physical Unclonable Functions (PUFs) are (believed to be) tamper-resistant primitives that can be integrated into attestation protocols to mitigate physical attacks. PUFs are based on the variations inherently present in different hardware components of a computing device, such as memory. PUFs can be used for device identification and authentication, or as a seed for random number generation. Uniqueness of components, upon which PUFs are constructed, comes from variations in the manufacturing processes, which are assumed not to be controllable by the adversary $\mathcal{ADV}$, and thus are not cloneable. Therefore, an on-board PUF can thwart a physical attack that aims to clone a compromised device. Also, since components used for PUFs,

such as on-chip Static Random Access Memory (SRAM), are anyhow part of the device (regardless of their use as PUFs) additional costs of PUFs are minimal. Several approaches to PUF-based attestation have been proposed [348, 230]. However, all PUF-based attestation schemes impose additional requirements on the device. Furthermore, recent work on PUF security demonstrated that conjectured security of certain PUF families, such as Arbiter PUFs, does not hold in practice [261], specifically, their unpredictability and uncloneability properties.

Double Verification. Assuming that the adversary $\mathcal{ADV}$ can compromise and control only a single device of the entire swarm *double verification* secures SEDA's attestation and aggregation. With double verification the integrity of each device D_i has to be validated by two other devices. In particular, the parent device verifies D_i's integrity (as in the standard SEDA scheme), and additionally, the grandparent in the spanning tree will verify D_i's integrity. This idea was also used by Jadia and Mathuria [211] with the goal to make hop-by-hop aggregation secure against single compromised nodes.

For double verification, SEDA's join protocol is extended such that each device D_i shares a symmetric key k_i with every direct neighbor (one-hop) as well as every two-hop neighbor. Hence, a single fully compromised device is unable to forge the attestation responses of its child devices. This approach remains secure even if more than one device is compromised as long as no two compromised devices are neighbors. However, the double verification approach induces extra computation and communication cost.

Absence Detection. The core idea of *absence detection* is that the devices of the swarm monitor each other with the goal to notice events that could indicate a physical attack. Prior literature on WSN security introduced the assumption that $\mathcal{ADV}$ has to take a device out of the swarm for a certain amount of time to perform a physical attack [104], e. g., to disassemble the device in order to extract its secrets. Thus, the absence of a device could indicate that this device is the target of a physical attack. There might be other reasons why a device is absent from the swarm, e. g., a faulty device might become inoperative. However, correct devices should never be absent, as SEDA assumes the swarm to be always connected (cf. Section 6.2.1). Therefore, by periodically verifying that a device is present an adversary $\mathcal{ADV}$ cannot perform a physical attack without being noticed, if the period for checking is shorter than the minimum time $\mathcal{ADV}$ needs to perform its attack.

For static swarms, each device D_i can detect the presence of its neighbor devices D_j by running a periodic check, i. e., requesting a heartbeat signal from its neighbors D_j. If no heartbeat signal is received from a device D_j it is considered absent. Using SEDA's shared attestation keys the device can authenticate each other preventing $\mathcal{ADV}$ from impersonating one device with another one. When D_i detects that one of its neighbors D_j has disappeared the entire swarm is informed by broadcasting a message identifying D_j as absent, and thus, potentially compromised device.

However, detecting absent devices in dynamic swarm is hard since the swarm's topology can change unpredictably, i. e., neighboring devices might disappear due to a topology change. With knowledge about the minimal time $\mathcal{ADV}$ needs to disconnect a

victim device from the swarm and a loosely synchronized clock among all devices of the swarm SEDA can be extended in two ways to address this problem.[5]

The first option is to execute SEDA's attestation periodically within the swarm, i. e., one of the swarm devices will be selected as initiator D_1. The initiator is chosen deterministically by all devices, e. g., in a round-robin fashion. Absent devices are then detected by using the previously described SEDA extension, which identifies devices that could not be attested correctly. Attestation for non-responding devices is considered unsuccessful, hence, all absent devices will be included in the accumulated report received by D_1. The result, i. e., the list of devices that could not be attested, will be announced by D_1 to the entire network.[6]

As a second option, all devices of the swarm can periodically announce their presence to the network, by sending out an authenticated timestamp. The announcements are broadcasted within the network, i. e., each device forwards previously unseen announcements to its neighbors, leading to an unfavorable message complexity of $\mathcal{O}(s), s = |\mathcal{S}|$. All devices maintain a list of received announcements and compare it against a master list of all devices in the swarm. Devices that did not announce themselves are considered absent, and thus, potentially compromised.

Implications. Absent detection requires knowledge about the swarm's devices, i. e., which devices are expected to be present. Thus, introducing new devices into the swarm requires that all devices have to learn about the new device. The join protocol needs to be extended to distribute the information about a new swarm member to all devices in the swarm.

6.2.8 *Conclusion*

In this section we presented SEDA, the first efficient attestation protocol for large groups of connected devices, or swarms. SEDA facilitates the highly scalable attestation of swarms with dynamic topology, consisting of heterogeneous devices. SEDA's advantages include: (1) reduction of the attestation protocol run time with (2) constant cost for the verifier, and (3) low and uniformly distributed cost for the swarm's devices. We showed SEDA's feasibility for low-end embedded systems by implementing it on SMART [139] and TrustLite [226], two security architectures for embedded devices.

We discussed different variants and extensions of SEDA, showing that it can be used, e. g., to identify compromised devices in the swarm, can be transformed into a probabilistic verification scheme, and can even be secured against strong adversaries compromising individual swarm devices via physical attacks.

5 These approaches are discussed in more detail in our conference publication [34] and were further developed in follow-up work DARPA [199].

6 In case the initiator device D_1 is compromised the final announcement might be forged. To detect a compromised initiator the heartbeat frequency must be at least double the minimum time assumed for a physical attack. This ensures that a potential compromised initiator is detected by the previous or subsequent round of self-attestation.

6.3 DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems

Embedded systems have been omnipresent for many years mostly performing simple tasks in isolation. However, emerging applications such as Internet of Things (IoT) (e. g., smart cities/homes/factories) and autonomous systems (e. g., cars, drones, or robots) require embedded systems to be highly connected and carry out *autonomous* as well as *collaborative* tasks. In autonomous collaborative systems no central entity coordinates actions of the individual (autonomous) devices. The involved devices interact with each other and coordinate their actions by exchanging information, such as sensor data, status information, and commands. The correct and secure functioning of an autonomous collaborative system strongly relies on the integrity of the devices involved in its operation. In particular, it must be ensured that exchanged data (sensor, commands, status) is correct and has not been *maliciously* modified on the originating device before it is sent to another device.

Remote Attestation (RA) is a powerful security service for verifying the integrity of remote device's software state. However, conventional RA solutions are either static or not scalable, and therefore, they are not suitable for autonomous collaborative systems.

Static attestation approaches provide a proof that the software initially loaded by a prover device is unmodified [346, 236, 139]. *Control-flow* attestation [3, 129, 416, 130], in contrast, enables a verifier to detect run-time attacks, such as code reuse attacks [328]. However, the strong assumption on the verifier's capability to distinguish correct and incorrect control flows of the prover limits the applicability of control-flow attestation for autonomous systems. Existing control-flow attestation solutions assume a powerful verifier that has a complete, and therefore very large, database of all valid execution paths that the verifier can quickly search to decide whether a reported execution path is valid. This is, with regard to storage and computation cost, very expensive for autonomous systems where each embedded device must be able to act as both verifier *and* prover.

Enforcement techniques for control-flow correctness, such as Control-Flow Integrity (CFI) [1, 2], do not provide information about the executed control-flow path to collaborating devices; they enforce static reaction policies, typically fail-stop, when a control-flow deviation is detected [1]. In contrast, control-flow attestation enables the verifier device to determine the appropriate reaction in case of an attack, i. e., allowing for more considered, contextual reactions policies.

The other limitation of previous attestation solutions, in particular for control-flow attestation schemes, lies in their limited scalability, i. e., they only allow the attestation of individual devices [346, 236, 139, 3, 129, 416, 130]. This problem was first addressed by SEDA (cf. Section 6.2). Following *collective attestation* schemes for networks of connected devices extended SEDA for different scenarios and assumptions [16, 199]. However, these schemes mainly assume a central verifier, which is not available in autonomous networks where the verifiers are distributed. Furthermore, these existing collective attestation schemes are not designed for control-flow attestation schemes.

Goals and Contributions. The goal of this work is the design of the first efficient and secure collective *run-time attestation* scheme for autonomous collaborative systems of embedded devices. Our scheme, called DIAT, ensures on-device data integrity against malicious manipulation. In the context of Data-Flow Integrity (DFI), the term data integrity is used to describe that *all* operations on data and variables obey a program's Data-Flow Graph (DFG). In contrast, in this work we focus on data that has been explicitly selected to be guarded and monitored. This data inherits its integrity guarantees from the integrity of the software modules that processed it. We provide a more detailed comparison to related work on data integrity in Section 6.4.

DIAT achieves efficiency by decomposing each device's embedded software into small interacting software *modules* and attest the control flow of those modules that are relevant for data exchanged in a given interaction between devices of an autonomous system.[7] DIAT's control-flow attestation guarantees that the data is only processed following a benign execution path, i. e., not altered maliciously.

In summary, DIAT provides end-to-end data integrity protection for collaborative autonomous networks combining the following contributions:

- *Data integrity attestation.* DIAT enables secure device interaction by ensuring the integrity of data shared between devices (e. g., sensor readings). This is done by linking data with an attestation report reflecting the correct generation and processing of the data (given our adversary model Section 6.3.1).
- *Modular attestation.* DIAT provides the design and implementation of modular attestation: The prover devices' software is decomposed into simple interacting modules and only those modules that process data of interest are attested. Modular decomposition reduces the attestation overhead for both, prover and verifier.
- *Novel execution-path representation.* DIAT's compact execution-path representation allows control-flow attestation of complex software programs with linear overhead.
- *Implementation and evaluation.* We demonstrate DIAT's effectiveness and efficiency with our implementation and evaluation of DIAT on collaborating autonomous drones. Our proof-of-concept implementation is based on the widely used Pixhawk PX4 flight controller [270].

6.3.1 *Model and Assumptions*

We consider *collaborative autonomous systems*, i. e., networks of connected entities (devices $D_i, i \in \{1, \ldots, n\}$) that interact with each other to perform one or more tasks. In such a network, typically no central entity is needed to coordinate actions of individual devices. However, a central entity may be used for maintenance reasons. The devices D_i within the network are mutually distrusting. Redundancy in the network allows the overall

7 For example, consider two drones that are interacting by exchanging their location information (e. g., GPS coordinates). The software modules to be attested are those responsible for determining and possibly processing (altering) a drone's GPS coordinates.

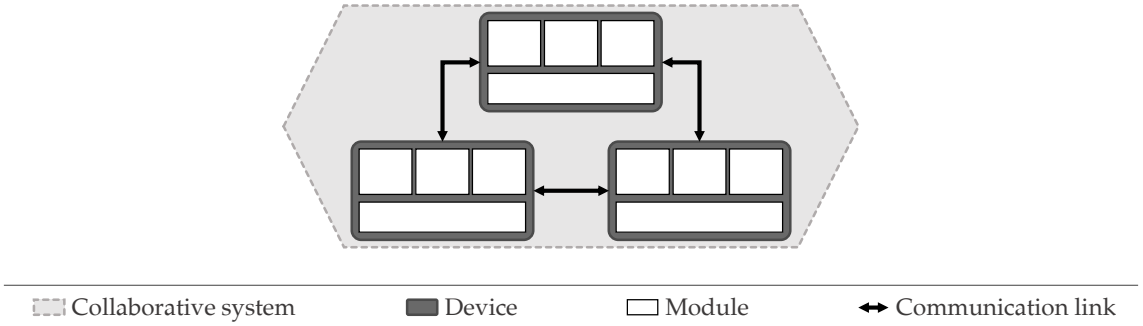


Figure 36: Abstract view of a collaborative autonomous system.

system to tolerate the misbehavior of individual D_i that can be due to faults or by attacks on devices. Redundancy is achieved, for instance, when the network is formed of homogeneous devices, so that one device D_i can easily be replaced by another device D_j in the network. In this work, for brevity, we assume *homogeneous* networks. The focus of this work is on how to detect the misbehavior of a device D in a collaborative autonomous system. How to react to an infected D is a complementary problem and depends on the underlying security policy.

Devices. Each individual D_i is self-contained and autonomous and can perform basic tasks by itself. Further, in order to coordinate their actions while performing more complex tasks, D_i exchange information, such as sensor data, status information, and commands. The software stack of embedded devices is less complex than that of typical desktop or server systems. However, this does not preclude devices with embedded Operating Systems (OSs) that run multiple tasks in parallel. Figure 36 illustrates the software model we assume in this work. Individual tasks or *software modules* $M_i (i \in \{1, \ldots, k\})$ of a device are strongly isolated from all other software components, including OS and other privileged software, utilizing a lightweight embedded hardware security architecture (see Section 6.3.6). The communication between software modules takes place over a well-defined interface which allows tracking of data-flow between modules M_i. All modules' software has to fulfill the requirements posed by the run-time attestation scheme that is part of DIAT. Modules' code must be instrumented such that all control-flow events are captured completely and correctly.[8] Our implementation on a popular flight controller for drones shows that our assumptions are realistic for the class of devices that DIAT is designed for.

Communication. Devices D_i are connected through wireless network technology, such as WiFi, Bluetooth or some custom solution, where each device D_i can be uniquely addressed. Communication does not need to be direct, i. e., devices D_i could, for example, form a meshed network to distribute messages.

8 DIAT can be combined with hardware-based run-time attestation solution, such as LiteHAX [130], removing the requirement for code instrumentation.

6.3.1.1 *Adversary Model*

We assume the adversary $\mathcal{ADV}$ has compromised and gained control over a subset of devices in a collaborative system. $\mathcal{ADV}$'s goal is to manipulate collaborative tasks by sending manipulated data to other, un-compromised autonomous devices.

We assume a stealthy adversary that aims to undermine the correct behavior of autonomous devices while evade detection. Therefore, Denial-of-Service (DoS) attacks, e. g., jamming the network communication between devices or trying to destroy devices (e. g., one device physically attacking/crashing into another device in case of drones or vehicles), are out of the scope of this work.

As common in RA literature, we consider software-only attacks. However, unlike conventional attestation schemes, we assume that $\mathcal{ADV}$ can manipulate code, e. g., while stored on persistent storage, as well as launch run-time attacks. Run-time attacks can be divided into: (1) *control-data* attacks, which introduce non-existing edges to a software's execution path that are not part of its Control-Flow Graph (CFG), e. g., Return-Oriented Programming (ROP) attacks [328]; and (2) *non-control data* attacks. Non-control data attacks can be split into two sub-classes. (2a) Attacks that do not add new edges but have an observable effect on the control flow of execution, for instance an unexpected number of loop iterations; and (2b) attacks that do not change a software's executed control flow at all, e. g., by changing of variables used in the generation of data. We excluded attacks that do not change the control flow as they are subject to active research and no general detection policy is known at the time of writing. If an appropriate policy is developed in the future, DIAT can be adapted to utilize it to also cover such attacks.

Each device D_i of the network is equipped with a lightweight hardware security architecture, which protects its Trusted Computing Base (TCB). The TCB includes the software components responsible for control-flow monitoring and data-flow monitoring as well as the security architecture's trusted components, e. g., when using TyTAN the System-on-Chip (SoC) hardware and TyTAN's security services. All other software, including an optional OS, is assumed to be potentially compromised and therefore untrusted. The hardware security architecture further ensures isolation between D's software modules M_i. Hardware attacks are considered out of scope in this work. Embedded security architectures are highly integrated into the SoC designs and not easily attacked [416].

We assume that D's sensors and actuators are trusted and report benign readings and perform actions as instructed.[9] This excludes false data injection attacks such as spoofed GPS signals. However, the software and drivers controlling the sensors and actuators might be controlled by $\mathcal{ADV}$.

6.3.1.2 *Objectives and Requirements*

The main goal of DIAT is to enable efficient and secure interaction/collaboration of embedded devices D_i in an autonomous system. This concerns several objectives as follows:

9 Misreading sensors and misbehavior of actuators due to faults are an orthogonal problem and can be handled by fault tolerant designs.

O1 *Code integrity on devices:* Unintentional/malicious alternation of the code running on a device D_i can be detected.

O2 *Data integrity on devices:* Unintentional/malicious alternation of the data on a device D_i (before being sent out to other devices D_j) can be detected. This means data can only be modified in a non-malicious way. This is necessary because D_i do not only exchange raw data, such as sensor readings; most data are processed before sent out. For instance, when sending position information, the receiver expects coordinates instead of a set of timestamps sent out by the GPS satellites and received by the GPS driver module M_{GPS}.

O3 *Data integrity and authenticity during transportation:* Malicious alternation of the data when traversing from one device D_i to another device D_j must be detected.

Attestation schemes are used to ensure the code and data integrity on devices D_i, i. e., any manipulation can be detected by the verifying device. To capture the run-time behavior of the code, DIAT adopts the idea of control-flow attestation [3]. However, in the context of autonomous embedded systems the attestation scheme must have the following properties:

P1 *Attestation efficiency*: Attestation is applied only to the critical code, i. e., those modules $M_c (c \in \{1, \ldots, l\})$ that process data of interest. Attesting the entire software on a device D_i would also allow making statements about the correctness of the data being processed. However, this naïve approach induces a huge overhead as the code responsible for the processing of a specific piece of data is usually only a small subset of the entire software stack.

P2 *Attestation latency:* The attestation should not cause delays beyond the bounds defined by the autonomous system's functional requirements. More concretely, the generation of an attestation report on the prover side must not delay the sending of data, while the verification on the verifier side must not delay the processing/usage of the received data.

To meet the above-mentioned objectives, each device D_i must fulfill the following requirements:

D-R1 *Isolation architecture:* The software modules/components $M_i (i \in \{1, \ldots, k\})$ are isolated from each other. Additionally, functional modules are isolated from privileged software components. Therefore, the (optional) OS is not part of DIAT's TCB.

D-R2 *Data-flow monitoring*: Software components/modules $M_c (c \in \{1, \ldots, l\})$ that access the data of interest d_s can be identified.

D-R3 *Control-flow monitoring:* The control flow of individual modules M_i must be captured when required.

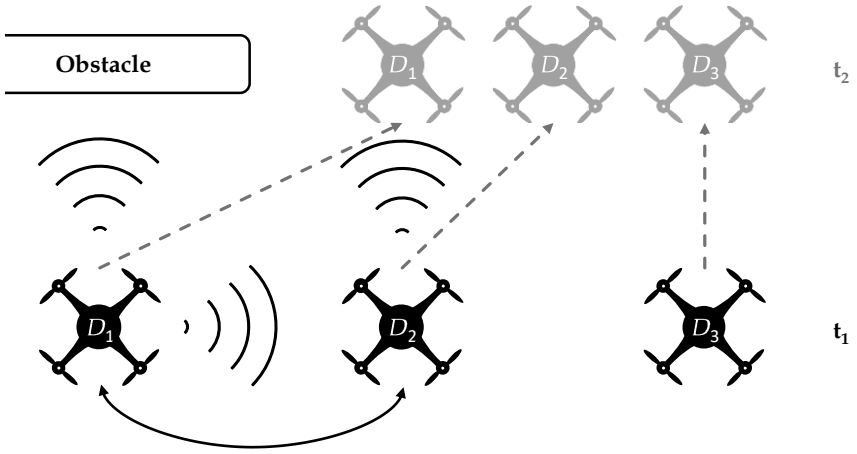


Figure 37: Example of collaborative drones. Each drone is an independent device D_i. Drones D_1 and D_2 coordinate their actions to avoid the obstacle without colliding.

D-R4 *Device key:* Each device D_i is equipped with a platform key-pair (sk_i/pk_i). D's secret key sk and other cryptographic parameters are protected by the hardware security architecture. It is exclusively accessible by D's TCB. Hence, the adversary $\mathcal{ADV}$ cannot forge the attestation reports generated by D's TCB or extract D's secret keys sk.

D-R5 *Attestor:* In a Remote Attestation (RA) protocol, the component(s) involved in measuring, attesting, and verifying a system's state are protected as part of the TCB.

Examples of lightweight security architecture that provide run-time isolation, secure storage, and secure boot (integrity) in order to protect DIAT's TCB have been developed by academia (e. g., TrustLite [226] and our real-time capable security architecture TyTAN described in Section 3.1) as well as industry (e. g., ARM TrustZone-M [30]). We discuss how these solutions can be adopted by DIAT in Section 6.3.6.

The following requirements are imposed on the communication between the individual devices D_i:

C-R1 *Secure channel*: The integrity and authenticity of data d sent over the network between the devices must be ensured. To enable the establishment of a secure channel each device must be equipped with a platform key-pair.

C-R2 *Infrastructure*: A Public Key Infrastructure (PKI) must be in place.

Use-case Example. We will now introduce an example use case of autonomous systems to illustrate our concept, which is simplified for the sake of clear illustration. The principle remains valid also for larger and more complex systems.

Consider a set of autonomously flying drones that collaborate to perform a distributed ground search, e. g., as part of a search and rescue mission. Each drone has a set of sensors, for instance cameras, which allow it to monitor a certain area at a time. In order

to cover more ground in a short time frame, multiple drones fly in formation. When the drones operate in close proximity, they have to coordinate with each other to avoid collisions. Figure 37 shows a scenario where a drone has to evade an obstacle.

The three drones (D_1, D_2, and D_3) are flying in line abreast. Each drone is an autonomous entity that can operate on its own. In particular, each drone is equipped with sensors that enable it to position itself (e. g., GPS), measure its movement (accelerometer, gyroscope, etc.), and sense its environment (e. g., via ultrasonic, infrared, Lidar). Note that those sensors are common in Commercial Off-The-Shelf (COTS) systems and even customer grade drones are equipped with such sensors. Finally, the drones are wirelessly connected to each other and can coordinate their actions.

During the drones' mission, D_1 detects an obstacle through a front-facing distance sensor. D_1 has three options for reacting to this situation: (1) it could abort its mission, i. e., stop or fly back; or it could circumnavigate by either (2) moving to the left, or (3) to the right. D_1 by itself does not have sufficient information to be able to decide in which direction to move, as the obstacle could expand in both directions. By exchanging information with other drones D_1 can learn that no obstacle has been detected in front of D_2. As a consequence, D_1 decides to evade the obstacle by moving to the right. Albeit this would prevent D_1 from colliding with the obstacle it would lead to a collision with D_2. Note that, D_1 is also aware of D_2's position through position sharing among the drones. Therefore, in order to avoid collision, D_1 has to coordinate with D_2. In particular, D_1 requests D_2 to move to the right, as well, in order to make space for D_1.

The result of the interaction between D_1 and D_2 is shown in Figure 37. The gray drones show their positions after they have coordinated their actions, where all three drones can pass by the obstacle safely.

6.3.2 *DIAT Design*

The core idea of DIAT is to enable autonomous devices D_i to trust the information they need to exchange in order to perform collaborative tasks within a network. This is done by means of a remote run-time attestation scheme that allows devices D_i to provide an authentic integrity proof of the data they exchange. Whenever information is exchanged, the sender D_S augments the data with a proof that this data has been generated and processed correctly. The receiver D_R can then verify this integrity proof, and thus, gain trust in the correctness of the received data.

Static (binary) attestation allows the verifier $\mathcal{VRF}$ to detect manipulations of the static code and data, e. g., due to a malware infection. However, static attestation cannot capture the run-time behavior of the code, and hence, cannot detect run-time attacks that leverage state-of-the-art code-reuse techniques, such as ROP [328]. To detect this class of attacks, previous works have taken the first steps towards run-time attestation schemes by means of control-flow attestation that records the execution path of the code running on the prover [3, 129]. Unfortunately, control-flow attestation schemes pose a significant overhead on both the prover $\mathcal{PRV}$ and the verifier $\mathcal{VRF}$ making it impractical for resource-constrained embedded systems. DIAT significantly reduces the overhead of

control-flow attestation and makes it applicable to collaborative systems of embedded devices.

6.3.2.1 *Challenges*

The main challenge of DIAT is to enable secure and efficient data integrity and control-flow attestation. Efficiency plays an important role given that collaborating devices are resource constrained embedded devices that must act simultaneously as both verifier and prover. As mentioned before, run-time attestation incurs significant overhead on the involved entities since: (a) as provers they need to continuously monitor the execution of their software, and (b) as verifiers, they need to know the benign execution paths and verify them at run-time.

The main contributions of DIAT tackle these challenges, enabling control-flow attestation for autonomous systems. In particular, (1) DIAT minimizes the size of the code to be attested using *modular attestation*, (2) it minimizes the attestation duration using *data-flow monitoring*, and (3) it *compresses attestation reports* using Multiset Hash (MSH) functions to represent execution path compactly.

6.3.2.2 *Architecture*

Figure 38 shows the abstract view of a device D using DIAT. To enable data integrity attestation, all modules $M_c(c \in \{1,\dots,l\})$ that process data of interest, i. e., sensitive data d_s, are attested while they are processing d_s. This reduces (1) the code to be attested to only those modules M_c that process d_s, and (2) the attestation time, i. e., modules M_c are only attested during the time frame in which they are processing the data. As shown in Figure 38, the data-flow monitor DFMonitor traces the flow of data within D_i and activates attestation for each module M_c that processes the data of interest. In Figure 38, data flows first from the sensor that created the data initially into module M_1, further into module M_2 and finally into module M_5 from where it is sent out. DFMonitor traces this flow of data and activates the control-flow monitor CFMonitor for the corresponding modules M_c (shown by the closed switches between CFMonitor and DFMonitor for the involved modules). We refer to data of interest as *sensitive data* d_s and to software modules involved in the processing of sensitive data as *critical tasks* or *critical modules* $M_c(c \in \{1,\dots,l\}) \subseteq M_i(i \in \{1,\dots,k\})$, which is a subset of all modules M_i of a device D_i.

The decision which data should be considered sensitive is application-dependent. In the scenario considered in this work (cf. Section 6.3.1) the location information exchanged between devices D_i is sensitive. However, for different operations or functionalities other data must be considered sensitive. This means it is not possible to determine a priori which modules M_i are critical modules M_c, hence, the identification of critical modules must be done dynamically at run time.

The control-flow information of critical modules M_c is recorded using an MSH function that captures the number of executions for every branch taken at run time. The details of our control-flow recording scheme are presented in our conference publication [4]. Using this execution path representation, the verifier learns, for instance, how many times a

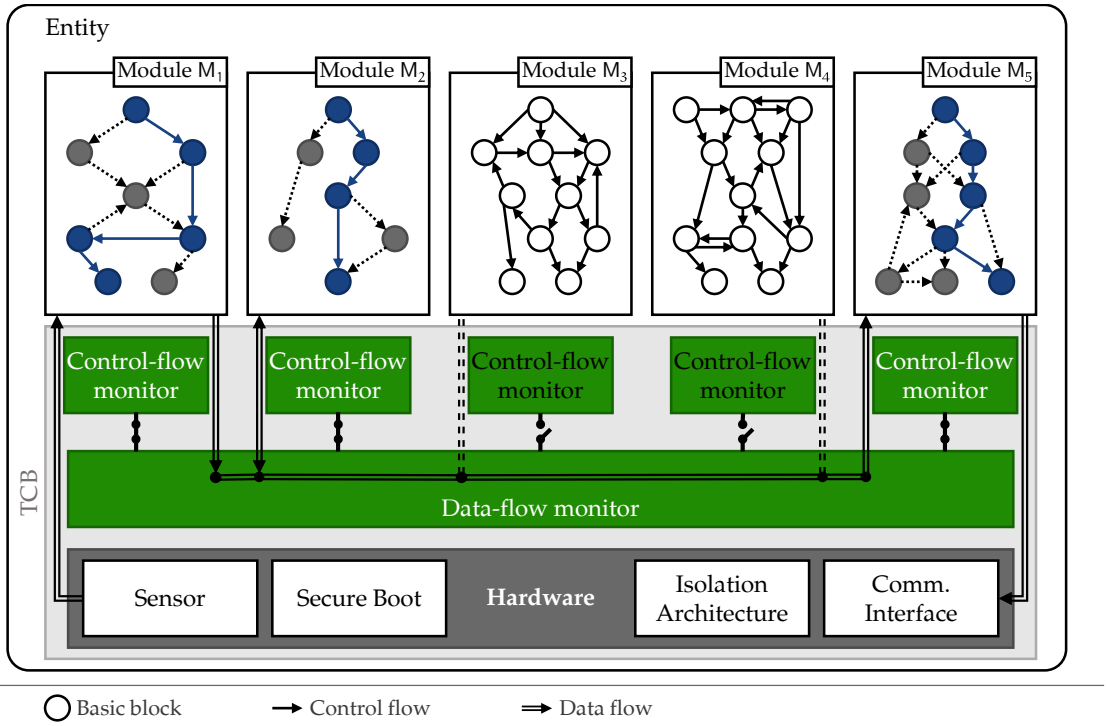


Figure 38: DIAT system architecture. Closed switches symbolize activation of control-flow monitoring. Data read from a sensor are passed through modules M_1, M_2 and M_5 as shown by the double-lined arrows. For those modules, the control-flow monitoring is activated while modules M_3 and M_4 are not monitored (symbolized by the open switches).

loop has been iterated through. This enables the detection of some classes of data-only attacks, e. g., those which increase the number of iterations of a loop (see Section 6.3.5). However, unlike previous control-flow attestation schemes [3], the verification overhead is significantly reduced, as the verifier $\mathcal{VRF}$ does not need to maintain and search a database, which includes all valid execution paths of software modules M_i. Subsequently, we explain each of our three building blocks in detail: (1) modular attestation, (2) data-flow monitoring, and (3) execution path representation.

Modular Attestation. The software on each D_i is decomposed into multiple small interacting software modules $M_i (i \in \{1, \ldots, k\})$ or tasks. Modular attestation is enabled by the fact that these M_i are isolated, i. e., the control-flow path recorded for a module M_i does indeed represent the behavior of M_i and independent of all other modules M_j. The module isolation is enforced by the underlying hardware security architecture, such as TyTAN [62] (cf. Section 3.1). We discuss DIAT's requirements on the hardware security architecture and possible instances of such architectures in Section 6.3.6. The isolation ensures that a module M_i's memory content as well as its execution cannot be influenced by other modules $M_j (i \neq j)$ or privileged software such as the OS.

DIAT does not impose strict requirements on the granularity of the software's modularization. Modules M are identified and delimitated using static analysis based on both, data and control-flow dependencies. DIAT does not require programmer's assistance

such as code annotations. Non-optimal module's, i. e., modules that could be further partitioned into sub-modules, lead to a decreased performance of DIAT, however DIAT's security is not affected. Many avionic and automotive systems (including flight controllers for drones) already have modular designs [270], which is highly important for these systems to be able to fulfill their safety requirements. Modular software design is a well-investigated topic in software engineering [14, 178, 13]. The cost of modular transformation is highly application dependent and may not be formulated in general statements. DIAT targets complex software with a modular design. For instance, the PX4 Autopilot software used in our DIAT prototype implementation (cf. Section 6.3.3) has such a modular software design where all modules communicate through a well-defined messaging system. Hence, no additional effort was required to transform the software in order to use DIAT. Small embedded software, e. g., controllers for sensors, may be monolithic. However, they are not the focus of DIAT.

Data-Flow Monitoring. Identifying the modules M_i involved in generating particular data is accomplished via coarse-grained data-flow monitoring. As modules M_i are isolated, they cannot directly communicate with each other (e. g., by accessing each other's memory). Communication occurs through a well-defined interface controlled and monitored by a component referred to as DFMonitor. This enables DIAT to trace the data-flow between modules M_i and identify critical modules M_c for particular sensitive data d_s dynamically at run time. Services for communication between tasks or modules M_i are common in most system architectures. Therefore, DIAT does not incur a system redesign. The communication service is just slightly extended to trace the flow of data. DFMonitor is described in more detail in Section 6.3.3.

Execution Path Representation. Existing control-flow attestation schemes [3, 129] induce a high overhead on the verifier, because the verifier must store and repeatedly search a very large database of *all* possible execution paths. The size of this database grows exponentially with the number of control-flow events in the code. To tackle this problem, we designed a novel execution path representation that is based on MSH functions [101]. Our representation provides an under-approximation of executed paths, i. e., the execution path cannot precisely be reconstructed on by the verifier. Nevertheless, the information contained in it is sufficient to detect all control-flow deviations, i. e., detect if a control-flow edge was executed that is not contained in the CFG. Furthermore, Data-oriented Programming (DOP) attacks that lead to the execution of valid but unexpected control-flow edges (e. g., number of loop iterations does not match the verifier's expectations) can be detected. We elaborate on DIAT's capabilities to detect DOP attacks in Section 6.3.5. Ultimately the verification policy applied by the verifier determines which attacks will be detected.

Summary. DIAT provides integrity guarantees for sensitive data d_s by linking exchanged data with a run-time attestation report of *all* software modules M_c that processed d_s. For all sensitive data d_{s_j} DIAT identifies and monitors *all* software components M_{c_j} that process a particular piece of sensitive data after it is initially received, e. g., from a sensor (sensor inputs are considered benign). If all modules M_{c_j} that accessed a piece of data d_{s_j} processed it only in benign ways, the integrity of d_{s_j} is preserved. Unauthorized

data modifications, i. e., by non-monitored software components or external entities, are prevented by module isolation and secure channels.

6.3.3 *Implementation*

We implemented DIAT for our use case of autonomously drones that collaboratively perform a distributed search, as describe in Section 6.3.1.2. The drones are controlled by an embedded system hardware platform, called PixHawk: an open-source flight controller used in many commercial drones. It is based on an ARM processor.

The PixHawk platform supports different autopilot software implementations.[10] We used the autopilot software maintained by the developers of PixHawk, called PX4, which is widely used in academic and industrial projects and is designed for resource-constrained autonomous aircrafts.

Our DIAT implementation leverages the software design of PX4 that consists of two layers, the *flight stack* and the *middleware*. The flight stack is designed as a set of software modules M_{fs}, providing the functionalities needed to control the aircraft. The middleware supports and composes the flight stack's software modules M_{fs}, e. g., by providing communication capabilities between modules and to external entities.

PX4 was extended with two main components, a data-flow monitor DFMonitor and a control-flow monitor CFMonitor.

DFMonitor. In PX4 all communication between software modules M_i is handled by a message broker. We extended the message broker to keep track of the data-flow of sensitive data d_s between modules M_i. DFMonitor is responsible for enabling the control-flow monitoring for software modules M_c whenever they process sensitive data d_s.

CFMonitor. For each software module M_i the control-flow can be tracked by DIAT. We instrumented all branch instructions of PX4 software modules, such that each branch taken can be recorded by the CFMonitor – when monitoring is active for a module M_j. When active CFMonitor records all taken (indirect) branches of a module M_j in an MSH-value.

Before sensitive data d_s is send to the verifier $\mathcal{VRF}$ d_s is extended with the MSH-values of all modules $M_c (c \in \{1, \ldots, l\})$ that processed it. The resulting message, i. e., d_s including its integrity report that contains $\text{MSH}(M_1), \ldots, \text{MSH}(M_l)$, is authenticated and integrity protected using a digital signature.

A detailed description of our DIAT implementation, elaborating on all technical challenges we tackled, is provided in our conference publication [4].

6.3.4 *Performance Evaluation*

We evaluated DIAT based on our implementation outlined in Section 6.3.3. The individual aspects of DIAT were evaluated separately, i. e., the performance of MSH and the attestation cost per software module. Additionally, we evaluated the characteristics

10 An autopilot is a software that is responsible for controlling an aircraft and keeping it stable.

of DIAT in our use-case scenario (cf. Section 6.3.1.2) and simulated large networks of collaborating devices to show DIAT's performance for different network typologies.

Our evaluation shows that DIAT can significantly reduce the size and complexity of control-flow attestation between devices. It can reduce the amount of software components that need to be recorded and attested in our use-case scenario by 95 %. The MSH representation of the control-flow information, additionally, reduces the attestation report's size, making it linear in the size of edges in the software's CFG. Compared to the attestation reports of previous control-flow attestation schemes [3, 129] this amounts to a reduction in size of approximately 98 %.

A detailed discussion of DIAT's performance evaluation as well as our measurements are provided in our conference publication [4].

6.3.5 *Security Analysis*

DIAT ensures the integrity of data exchanged in a collaborative autonomous system. In this section, we will show that DIAT fulfills all security requirements identified in Section 6.3.1.2. An adversary aiming to violate DIAT's security guarantees can manipulate sensitive data in three different phases: (1) while the data is generated or initially sensed by the platform's hardware, (2) while the data is processed on the platform, and (3) when the data is transferred to the verifier.

6.3.5.1 *Off-Device Security*

DIAT relies on the correctness of the initial data d, spurious sensor data constitute an orthogonal problem. While sensitive data d_s is transferred to the verifier $\mathcal{VRF}$, d_s is protected by cryptographic means, i. e., d_s is digitally signed using a key only accessible to the prover $\mathcal{PRV}$'s TCB. Hence, the adversary $\mathcal{ADV}$ cannot manipulate d_s without being detected as $\mathcal{ADV}$ cannot generate a valid signature for altered d_s. This means that DIAT fulfills the security requirement for *data integrity and authentication during transportation*, as identified in Section 6.3.1.2.

6.3.5.2 *On-Device Security*

To ensure data integrity on the platform itself, DIAT has to counter a number of attack vectors and strategies. $\mathcal{ADV}$ can target (1) DFMonitor or CFMonitor, (2) a module M_i that is *not* processing sensitive data d_s, and (3) a module M_c that is processing d_s.[11] (4) The adversary can aim to exploit dynamic data dependencies and try to inject "untrusted" data into the processing of sensitive data d_s.

Control-flow and Data-flow Monitor. CFMonitor and DFMonitor constitute DIAT's TCB, which is assumed to be immune to attacks, as described in Section 6.3.1.1. DIAT's platform security architecture (Section 6.3.6) ensures the isolation of CFMonitor and DFMonitor as well as their initial integrity by means of secure boot.

11 For sake of clarity we concentrate on an adversary targeting a single module, extending the security arguments to multiple modules is straight forward.

Non-critical Modules. Software modules M_i that are not relevant for the integrity of sensitive data d_s are, by definition, non-critical modules. Therefore, any compromise of a non-critical module does not give $\mathcal{ADV}$ any advantage towards compromising the integrity of d_s.

Critical Modules. Software modules M_c that can influence the integrity of d_s are considered critical modules. They can be subjected to a number of attacks:

Code Integrity. $\mathcal{ADV}$ can manipulate the code of M_c either before the module is loaded or at run time. Manipulation of M_c before load-time is detected by static attestation that is performed for every module when it is loaded. At run time, M_c's code is protected by the isolation that is enforced by the used security architecture. Therefore, $\mathcal{ADV}$ can only manipulate the code from *within* M_c, i. e., by techniques such as code injection. However, these attacks are prevented by Data Execution Prevention (DEP), which eliminates the possibility to insert new code as well as the option to alter or overwrite existing code. Hence, the code integrity of DIAT's modules is ensured and the requirement for *code integrity on device* is fulfilled.

Module Data Integrity. The integrity of a module M_i's data is crucial for the correct operation of M_i, i. e., that M_i is processing sensitive input data d_s correctly.

Different types of run-time attacks modify different data and have different effects on the control flow as described in Section 6.3.1.1. Control-data attacks, such as ROP [328], directly influence the control flow of a module's code and introduce new control-flow edges in the executed control-flow path. DIAT's CFMonitor captures *all* control-flow transitions executed in all $M_c (c \in \{1, \ldots, l\})$ while processing sensitive data d_s. This information is provided to $\mathcal{VRF}$ that can easily check whether all executed transitions – independent of the order in which they were executed – are legitimate, by checking if they are contained in M_c's CFG.

Non-control data attacks do not introduce control-flow edges outside of the CFG in a module's execution path. The class of non-control-data attacks can be divided into two sub-classes, (a) attacks that influence a module's execution path [98, 196], and (b) attacks that leave the execution path completely unchanged.

Existing DOP attacks, which fall into the first sub-class, are often target-specific and aim to achieve the execution of program functionality that is restricted, i. e., functionality that should not be available in the given execution context [98]. DIAT can detect these types of attacks, given sufficient context information on the verifier side. In particular, DIAT's MSH representation does reveal to $\mathcal{VRF}$ whether a restricted control-flow path was executed in a module M_c. More general are DOP attacks that provide the adversary full control over a target program's operation, i. e., DOP attacks achieving Turing-completeness [196]. These attacks combine multiple instruction sequences, called data-oriented gadgets, which are chained together using a dispatcher gadget. The dispatcher gadget is a loop (or loop-like construction) that has to be iterated through repetitively. This unexpected count of iterations can be detected by $\mathcal{VRF}$ in DIAT's control-flow report, despite the loss of order information due to its constant size MSH representation.

DOP attacks from the second sub-class neither execute an unexpected path in M_c's CFG nor execute transitions (e. g., of loops) atypically often. These types of attacks, e. g.,

attacks that modify the values of variables that are used in calculations, are currently not detectable with DIAT. In general, Data-oriented Programming attacks are subject to active research and no generic detection policy for these attacks existed at the time of this work's writing. However, DIAT can be adapted and extended with new detection policies on the verifier side when they are developed in the future.

Data Dependencies. The integrity of sensitive data d_s is dependent on the operation applied to it, i. e., the correct operation of the modules $M_c (c \in \{1, \ldots, l\})$ processing it, as well as other data involved in its processing. An adversary $\mathcal{ADV}$ could aim at manipulating data that is eventually integrated into d_s. In general, DIAT uses dynamic control-flow tracing to determine which modules M_j provide input to the processing of d_s and considers those input data d_j sensitive as well.

In particular, if some module M_c processes sensitive data d_s and receives or requests input data d_j, this input is considered sensitive. Therefore, other modules $M_j (j \in \{1, \ldots, o\})$ providing such input data d_j have to be monitored as well. If M_j produce the input data d_j "on-demand", their monitoring can be activated before the sensitive data d_j is produced, i. e., the integrity of d_j can be verified. However, for data d_h that has been produced at an earlier point in time no information about the correct creation of d_h is available. DIAT addresses this problem by ensuring that all inputs d_h for generating d_s are generated on-demand, i. e., DIAT provides transitive verifiable data generation and processing.

In our prototype implementation, modules M_h that produce data on-demand would publish their results for use by other modules M_i. In case that a critical module M_c requires such data, the buffer of already published results is flushed, afterwards the producing module M_h is monitored while it is generating new data d_h. The M_c can then use the new results d_h, which were generated while the producing module M_h was monitored, i. e., the correctness of results is verifiable.

DIAT's transitive verifiability of data integrity fulfills the requirement for *data integrity on device* (cf. Section 6.3.1.2).

6.3.6 *Discussion*

DIAT's relies on a platform security architecture providing (1) secure storage, (2) support for secure boot, (3) isolation of software modules, and (4) secure Inter-Process Communication (IPC). These requirements are rather general and can be fulfilled by different security architectures. In this section, we detail on two security architectures and show that they fulfill DIAT's requirements. First, we discuss TyTAN, our security architecture for embedded real-time systems introduced in Section 3.1. Secondly, we show that TrustZone-M [30], an industry solution that is available in COTS devices, fulfills DIAT's requirements, as well.

6.3.6.1 *TyTAN*

Our security architecture TyTAN is described in detail in Section 3.1.

TyTAN fulfills DIAT's requirements on the security architecture as follows:

- *Secure storage:* TyTAN provides a security service known as sealing, which allows secure storage of data.
- *Secure boot:* TyTAN provides secure boot as part of its architecture.
- *Software module isolation:* Providing isolation of software modules is a main objective of TyTAN, it utilizes an Execution-Aware Memory Protection Unit (EA-MPU) to implement the isolation.
- *Secure IPC:* TyTAN provides a mechanism for secure IPC.

TyTAN Performance. TyTAN imposes minimal overhead on a system's performance, as we showed in our evaluation in Section 3.1.6. We evaluated TyTAN's overhead with regard to different system tasks, in particular, creating a secure task, measuring a task, and saving and restoring the context of a task. TyTAN incurs overhead when creating a new task, however, when the system is running, the only overhead of TyTAN originates from the secure context switch between isolated modules. Our evaluation shows that the overhead for secure context switches is minimal, i. e., one additional jump instruction plus erasing the module's content from the Central Processing Unit (CPU) registers.

6.3.6.2 *TrustZone-M*

The *Armv8-M Security Extensions* provide a security architecture commonly referred to as TrustZone-M [30]. It provides similar functionalities as the original TrustZone, i. e., partitioning a platform into two logical sub-systems – one for legacy software and a second sub-system for security critical functionalities (cf. Section 2.3.1). The postfix *M* – for Microcontroller – designates the target platform: processors from ARM's series of processors designed for deeply embedded systems, named Cortex-M. TrustZone-M is available starting from the 8-th generation of ARM Cortex-M processors Armv8-M.

With TrustZone, the SoC is divided in two worlds, called *secure state* or *non-secure state* in the context of TrustZone-M. System resources, e. g., memory sections, can be assigned to these worlds; the secure state has exclusive access to resources assigned to it.

TrustZone-M-enabled processors, additionally, distinguish two privilege levels, available in both, secure state and in non-secure state. The unprivileged *thread mode* runs application code. The privileged *handler mode* is responsible for exception handling and resource management.

The system always starts in secure state; hence, the secure state software has control over the platform before non-secure software is loaded. This enables it to setup protection mechanism, for instance configuring memory partitioning, before non-secure software gets executed. To isolate the internal state of the secure and non-secure software, TrustZone-M provides mechanism such as register and exception banking.

- *Secure storage:* From the platform key storage encryption keys for each software module can be derived and made only accessible to the corresponding software modules (e. g., per secure IPC). The platform key is only accessible to the trusted secure state (i. e., TCB). This enables software modules to securely store data.

- *Secure boot:* TrustZone-M provides secure boot for the secure state software, i. e., ensuring its initial integrity. The platform starts in the secure state; thus, the secure state software cannot be manipulated by non-secure state software initially. Before non-secure state software is loaded the isolation mechanism can be enabled to preserve the secure state software's integrity.
- *Software module isolation:* Isolation for software modules can be enforced by configuring the memory access control policy (stored in and enforced by the Memory Protection Unit (MPU)). To enable isolation between different software modules a small trusted management software, which it protected in the secure state, can update the access control policy on every context switch, similar to a typical OS enforcing process isolation.
- *Secure IPC:* Secure communication can be realized in two ways. (1) Shared secure memory between software modules can be configured by a small trusted management software, similar to the configuration of memory isolation for software modules (see above). (2) DIAT's DFMonitor, which is part of the TCB, can act as a secure IPC broker.

6.3.7 *Conclusion*

In this section we presented DIAT, a novel attestation approach that shifts focus from device integrity to data integrity. DIAT enables efficient run-time attestation and thus enables the mutual attestation of resource constrained embedded devices, in scenarios where devices need to act as both, prover and verifier. This is achieved by combining three new building blocks: (1) data integrity attestation, (2) modular attestation, and (3) MSH-based representation of execution paths. By composing these building blocks, DIAT allows the detection of compromised data shared between collaborating devices, which is highly relevant to enable reliable cooperation of devices. We demonstrated DIAT's applicability for a real-world use case for collaborative drones. We implemented and evaluated DIAT on a state-of-the-art flight controller for drones, i. e., an embedded system with strict real-time requirements, and demonstrated its applicability for these types of systems.

6.4 Related Work

Remote Attestation (RA) is a security service that has been extensively studied in the past, which led to a large variety of solutions that are related to the solutions presented in this work. Integrity enforcement is an alternative approach investigated in literature (cf. Section 6.4.2).

6.4.1 *Attestation*

The goal of RA is to ensure the integrity of a remote device, or in the case of collective attestation to verify the integrity of a set of devices. Depending on the information collected and integrated into an attestation report, the verifier can verify a device's integrity with respect to different types of attacks. Static attestation allows the verifier to detect manipulation of a device's static memory content, in particular its program code. Run-time attestation provides the verifier information about a device's run-time behavior.

6.4.1.1 *Static Attestation*

The concept of static attestation allows a verifier to verify the software configuration of a prover system, i. e., the load-time software configuration of a platform or its components, based on a status report generated in an authentic and integer way on the prover device. Existing approaches can be divided into three categories based on the mechanisms and components used for authenticating attestation reports, discussed individually below. Software-based attestation does not use cryptographic secrets to authenticate attestation report, in secure hardware-based approaches the authentication secret is managed by dedicated hardware modules, and in hybrid approaches hardware-protected, trusted software has access to the prover's authentication secret and is responsible for authenticating attestation reports.

All solutions that describe verifier authentication [65] and were presented before our work have in common that a malicious verifier is not considered. Following works developed new approaches to tackle this problem, e. g., by using non-interactive attestation schemes [200, 83, 86].

Software-Based Attestation. Many software-based attestation techniques have been proposed in the past. One early example is Pioneer [343]. It computes a checksum of a prover device's memory using a function with side-effects in its computation, e. g., including status registers in the calculation, such that any emulation of this function incurs an additional timing overhead (delay) that is sufficient to detect manipulations. Also, Kennell and Jamieson [217] as well as Gardner et al. [153] rely on architectural side effects to ensure the integrity of the measurement method. Attestation that relies on time-based checksums has also been adapted to embedded devices in [344, 342, 221, 243, 244, 346]. However, some basic assumptions that underlie these techniques are uncertain [349]

and several attacks[12] on software-based attestation schemes have been demonstrated, e. g., Castelluccia et al. [88], Kovah et al. [236].

In general, all current software-only techniques rely on strong assumptions about adversarial capabilities, such as the adversary being passive while the attestation protocol is executed and optimality of the attestation algorithm and its implementation. While applicable to very specific settings, e. g., attestation of computer peripherals, this general approach is not viable for attestation performed over a network as the required preconditions are hard to achieve in practice [32]. Gligor and Woo [165] aim to overcome the constrains of previous software-based attestation approaches. They formalize software-based attestation for an abstract computer model (cWRAM – Word Random Access Machine) and show that in this model an optimal measurement function can be constructed. While they show that an adversary limited to the resources of the prover device, which complies to the cWRAM model, cannot manipulate the measurement without increasing the execution time, the authenticity of the prover device cannot be guaranteed limiting this approach to local verification settings. Gligor [164] shows how to maintain trust in a system using software-based attestation [165].

Secure Hardware-based Attestation. Trusted Platform Modules (TPMs) [382] are present in many modern commercial systems, from servers to laptops. A TPM can store an integrity checksum computed over the memory at boot time in protected memory called Platform Configuration Registers (PCRs). The stored checksum can be sent to a remote verifier for validation. TPMs can also protect data against a compromised Operating System (OS), e. g., protect encryption keys from misuse. The Root of Trust (RoT) is the TPM plus the Basic Input Output System (BIOS) that performs the first measurement of software upon boot. Several concrete architectures have been proposed that rely on a TPM as a foundation [307, 221].

Datta et al. [121] present a logic for secure systems. They use it to describe attestation protocols standardized by the Trusted Computing Group (TCG), without providing a definition of attestation [121] and relying on the presence of a secure TPM device.

Dynamic Root of Trust for Measurement (DRTM) is an extended mechanism added to the TPM specifications in version 1.2 [382]. It has been implemented by major vendors, e. g., Intel Trusted Execution Technology (TXT) [204] and AMD Secure Virtual Machine (SVM) [9]. Basically, Dynamic RoT is a way to perform attestation *dynamically*, i. e., after boot. This is accomplished by allowing a specific Central Processing Unit (CPU) instruction to reset the state of some PCR, isolate a memory region, hash and atomically execute its content. Flicker [264] is an architecture that establishes DRTM on commodity computers. It takes advantage of Intel TXT and AMD SVM by executing a piece of application logic (PAL) on the prover. This architecture was extended by TrustVisor [266]. It provides a dynamic root of trust for PALs using a minimal hypervisor. TrustVisor significantly improves performance of the DRTM primitive. There are several other proposals that deal with establishment of trust on remote systems [217, 301, 265, 413]. Underlying platforms range from Web servers to embedded systems.

12 For example, vulnerabilities to Time-Of-Check-Time-Of-Use (TOCTOU) attacks identified by Kovah et al. [236].

Hybrid Techniques. SPM is a hardware-based mechanism for process isolation [368]. It relies on a special *vault* module, bootstrapped from a static RoT. This vault bootstraps SPM-protected programs, which gain exclusive control over the protection of their own memory sections. Another proposal from 2011 is SMART [139] – a hardware/software-co-design-based scheme for establishing a dynamic RoT in embedded devices. Its focus is on low-end Microcontroller Units (MCUs) that lack sophisticated features, such as specialized memory management or TPMs. SMART requires no additional hardware – only a few small changes to the MCUs. SPM and SMART share some key features, such as the use of program counters to restrict access to secret key storage, and code entry point enforcement. However, unlike SMART, SPM does not provide a dynamic RoT. It also involves a larger Trusted Computing Base (TCB) and is generally oriented towards higher-end embedded systems with a Memory Management Unit (MMU) or a Memory Protection Unit (MPU). Furthermore, SPM requires the addition of custom instructions to the core.

A refinement of SMART was presented in [147], where more precise specification of minimal architectural features needed to support attestation were derived.

Another follow-on is the TrustLite architecture for embedded systems [226]. It enables running arbitrary code, called trustlets, isolated from the rest of the system. An Execution-Aware Memory Protection Unit (EA-MPU) ensures that the data of a trustlet can be accessed only by the code of the trustlet to which the data belongs. Furthermore, EA-MPU can be used to control access to hardware components such as peripherals. Authenticity and confidentiality of the code and data of trustlets is ensured by means of secure boot. The main difference to SMART is that the memory access control rules of the EA-MPU in TrustLite can be programmed as required by trustlets. In contrast, memory access control rules of SMART are static. Also, TrustLite supports interrupt handling for trustlets, while the security-critical code in Read-Only Memory (ROM) of SMART must not be interrupted during execution.

Attestation Formalization. ElDefrawy and Tsudik [138] discuss the set of requirements RA schemes need to fulfill in order to be secure. Furthermore, they propose the use of computer-aided proving in order to achieve strong security, safety and robustness guarantees for RA implementations.

VRASED provides the first formally verified implementation of a minimal RA architecture [127]. It proves soundness and security for a hybrid RA architecture combining software and hardware, similar to SMART [139]. Furthermore, VRASED provides a proof for an extended version of SMART supporting verifier authentication following the protocol introduced in Section 6.1. PURE extends VRASED with three provably secure services that leverage RA, providing provably secure software update, memory-erasure, and system-wide resets [309].

RA-related Works. ERASMUS addresses the same problem as discussed in Section 6.1, i. e., preventing the misuse of RA for Denial-of-Service (DoS) attacks against prover devices, using a different approach [85, 86]. Additionally, ERASMUS is concerned with the problem of capturing *mobile adversaries*, i. e., an adversary that leaves a prover device without leaving a trace, potentially before a device's software state is measured in a

RA protocol. The concept of *Quality-of-Attestation* is introduced to capture the timing aspects of RA. ERASMUS splits RA into two independent phases. (1) The measurement phase, during which the software state is captured. This phase is executed periodically (potentially at a high frequency to capture mobile adversary with high probability) and does not involve interaction with a verifier. (2) The collection phase, in which the verifier requests already created measurement reports. However, since not expensive operation, i. e., measurement of the software state, is triggered in the collection phase, a malicious verifier cannot disrupt the prover device.

Most attestation solutions assume temporal consistency of the memory while it is measured, e. g., by setting the memory immutable or by exclusively executing the measurement function uninterruptible. Temporal consistency is important, e. g., to counter self-relocating malware and mobile adversaries. However, this requirement easily contradicts with real-time requirements. Carpent et al. [83, 84] investigate different strategies when to lock and release memory during measurements and the resulting security and availability grantees. The different approaches were extensively evaluated by Carpent et al. [84]. Additionally, they investigated additional strategies for temporal consistency, e. g., monitoring the consistency rather than enforcing the consistency of memory.

TyTAN (cf. Section 3.1.4.3) prohibits tasks to run while they are measured, preventing self-modifying malware from relocating itself during the measurement process. Additionally, secure tasks are isolated preventing any other entity from modifying the memory of the measured tasks, hence, TyTAN provides temporal consistency for its secure tasks.

Kohnhäuser and Katzenbeisser [228] propose a secure software update scheme for networks of devices that relies on hardware security features available in low-end embedded devices, such as execute-only memory. In their scheme, devices mutually attest each other to validate the correct deployment of updates. In order to enforce the correctness of the overall network, devices that are found to be incorrect, e. g., if they have not correctly installed a software update or cannot provide an appropriate proof via RA, are excluded from the network. In particular, devices only interact with neighboring devices after verifying that they are in the correct state.

6.4.1.2 *Run-time Attestation*

Several schemes for control-flow attestation have been proposed [3, 129, 416, 130]: C-FLAT [3] enables a prover to attest the exact control-flow path of an executed program to a remote verifier. However, C-FLAT is not scalable and poses high verification overhead on the verifier. Hence, it cannot be simply combined with existing collective attestation techniques. To improve the performance of C-FLAT on the prover, LO-FAT [129] was developed. LO-FAT leverages hardware assistance to track control-flow events and performs hash calculations parallel to program execution. Moreover, LO-FAT supports control-flow attestation of legacy code since binary instrumentation is not required. Control-flow attestation schemes such as C-FLAT and LO-FAT induce high verification cost, and therefore, they are not applicable to autonomous systems. DIAT can leverage hardware assistance as described by LO-FAT to further reduce the overhead on the prover in an autonomous system.

ScaRR (Scalable Runtime Remote Attestation for Complex Systems) pursues the goal to make control-flow attestation applicable to complex systems, for instance in cloud environments [378]. To prevent state explosion, ScaRR divides control-flow traces into sub-traces, similar to C-FLAT, to handle, for instance, loops as dedicated sub-traces [3]. ScaRR introduces so-called checkpoints that mark the beginning/end of a sub-graph in a program, e. g., at loop entries/exits, exception handling methods, or start/end of threads. However, ScaRR needs to trace the *entire* software execution of a prover and report *all* sub-traces to the verifier. DIAT, in contrast, decomposes the prover's software into *isolated* software modules, hence only the traces of a subset of modules need to be reported to the verifier, actually reducing the overall amount of software to be measured for an informative attestation.

Koutroumpouchos et al. [235] describe an approach to build a lightweight control-flow attestation system by minimizing the scope of software on a system to be attested, called CFPA (Control-Flow Property-based Attestation). The proposed approach follows the same idea as DIAT and limits attestation to "critical" modules, thus reducing the overall cost. However, CFPA, unlike DIAT, does not allow the dynamic information-flow tracing between modules on the prover device, therefore, entire services have to be encapsulated into single modules. Furthermore, CFPA does not provide strong isolation between different software modules and does not provide isolation of software modules from privileged software, such as the OS, leading to a large and complex TCB comprising the system's entire privileged software components. CFPA performs run-time attestation of software modules periodically, DIAT, in contrast, attests modules while they process critical data, allowing DIAT to avoid TOCTOU problems.

IoTA presents a framework for run-time integrity verification of connected (cyber-physical) systems [103]. It aims to pose minimal requirements on resource-constrained edge devices, therefore, it neglects lightweight hardware-based security architectures, such as TyTAN (cf. Section 3.1), and relies on a trusted software layer to provide measurements and protect secret information.

OAT introduces the notion of "Operation Execution Integrity" (OEI) [372], addressing the path explosion problem of control-flow attestation. OAT's idea is to limit the attestation to individual operations of a prover device, i. e., tasks with defined entry and exit functions, thus minimizing the size and complexity of the software to be attested. While the core idea is similar to DIAT, which also attests tasks individually, OAT assumes a different setting in which the prover device is executing individual operations that are recorded and reported to the verifier. DIAT, in contrast, focuses on the data generated and processed by a prover device, which can be generated by a single task in a single operation or jointly by multiple tasks. In OAT, all tasks processing a critical data element must be considered as a single operation leading to large and complex operations for many realistic use cases. OAT considers prover devices without an OS and without multi-threading, i. e., a system without concurrent software execution. Interrupts, when occurring during an attested operation, are handled as separate operations that need to be attested as well. DIAT isolates all tasks protecting their integrity from all other, potential malicious software on the system. Hence, DIAT supports interruption of tasks, including

task scheduling by an OS as well as multi-threading systems, making it applicable to much wider range of system and use cases.

6.4.1.3 *Collective Attestation*

Park et al. [299] proposes multi-prover attestation for homogeneous systems. Their idea is that the verifier does not verify each individual attestation report, but just compares integrity measurements of multiple provers. In contrast, our attestation scheme supports a large number of provers running the same or different software and distributes verification of attestation reports over the whole swarm.

Following the development of SEDA, various swarm attestation schemes have been developed that improve and extend the basic concept of SEDA [16, 34, 199, 201, 82].

SANA uses an aggregate signature scheme to minimize the verification overhead on swarm devices [16]. Furthermore, swarm devices that are not attested, i. e., not act as prover, are not required to be equipped with a trust anchor in SANA.

DARPA extends SEDA to operate in a stronger adversary model considering physical attacks [199]. It is based on absence detection via periodic heart beats, unveiling devices that have been offline and therefore potentially physically compromised (cf. Section 6.2.7).

SCAPI improves on the idea of physical absents detection in swarms of networked devices [229]. SCAPI periodically updates a system wide session key such that only devices that received *all* session key updates can obtain the latest key. A device that had been taken out by an adversary to perform a physical attack will miss the distribution of at least one session key rendering it incapable of obtaining any of the subsequent distributed session keys. Hence, physically compromised devices cannot participate in the network's operation as they do not have the current session key. Furthermore, physically compromised devices can be identified by SCAPI during attestation because the successful completion of the attestation protocol requires each prover device to have the current session key.

SlimIoT adapts the idea of SCAPI and regularly updates a shared key (epoch key) among uncompromised devices to exclude compromised devices from the swarm [18]. In addition to detecting absent devices (which are considered physically attacked) SlimIoT additionally performs periodic software attestation, resulting in a list of non-compromised devices. New epoch keys are derived from a hash-chain managed by a verifier device and distributed to all non-compromised devices.

Carpent et al. [82] present lightweight swarm attestation schemes (in two variants) that extend SEDA with regard to verifier authentication as well as timing behavior of the attestation schemes. The first variant, called LISA-α, works asynchronously and makes minimal assumption on the underlying security architecture. The synchronous variant, called LISA-s, reduces communication overhead by aggregating attestation reports. However, LISA-s increases the protocol complexity, which translates to more complex implementation and increased code size for the trust anchor of the system. Additionally, the authors introduce a "QoSA: Quality of Swarm Attestation" metric to allow comparison of swarm attestation schemes.

MTRA describes a hierarchical attestation scheme, where devices equipped with TPMs serve as a proxy-verifier for devices without TPM [375]. SEDA does not restrict the roles of device in the network, unlike MTRA that only allows devices equipped with TPM to serve as proxy-verifiers. This allows SEDA to dynamically create a spanning tree determining the attestation relations between devices. Furthermore, SEDA requires only minimal security architectures for the swarm devices.

SHeLA (Scalable Heterogeneous Layered Attestation) is based on the assumption that all swarm devices can be attested by proxy-verifiers [319], which are more powerful compared to the swarm devices. The proxy-verifiers use single-device attestation protocols to attest each swarm device individually, the results of the different proxy-verifiers are synchronized between them to compose an attestation report reflecting the state of all swarm devices, which is provided to the central verifier. Compared to SEDA, this approach is limited in scalability due to its static two-layer hierarchy.

WISE divides the devices of a swarm into clusters, with the goal to attest devices that have a higher probability of being compromised more often, while other devices are attested less frequent leading to lower overall cost [17]. The verifier uses a Hidden Markov Model (HMM) to learn, based on past attestation results, which clusters need to be attested at what time. WISE uses swarm attestation within each sub-swarms or clusters of devices utilizing the scalability of the concept introduced in SEDA.

VL et al. [392] propose a swarm attestation scheme where each device in the swarm is verified by multiple other devices, called set-verifiers, with the goal to tolerate faulty devices. The set-verifier either use consensus to decide on the state of the prover device or all set-verifier report to the initiator device, i. e., the device initially approached by the external swarm verifier, which will make a decision based on the reports received from all set-verifiers, e. g., via majority vote. To improve scalability of the approach the authors propose to partition the swarm into sub-swarms for which a leader device collects all attestation results and forwards them to the initiator device. While fault tolerance was not a design goal of SEDA its concept can easily be adapted to tolerate faulty devices, i. e., the swarm verifier approaches different initiator device to run SEDA's attestation protocol. Each initiator device will initiate the creation of a new spanning tree, thus with high probability (assuming a well-connected network topology) a faulty device will not be the root of the same sub-tree in different spanning trees. This adaption of SEDA does not rely on a fault-free initiator device or sub-swarm leaders. This approach, furthermore, works in an on-demand fashion, i. e., only if the attestation fails due to potentially faulty device the swarm verifier has to repeat the attestation with another initiator device. Hence, the overhead due to redundant verification occurs only in case a faulty device exists.

DADS (Decentralized Attestation for Device Swarms) aims to provide a decentralized swarm attestation solution [404], building on top of SEDA. When a compromised device is detected during an attestation session, it is excluded from the swarm. A device (proxy-verifier) that detects the compromise replaces the compromised device in the attestation tree, i. e., it inherits the compromised device's children, which implies the assumption that all children of the compromised are reachable by the proxy-verifier device.

De Oliveira Nunes et al. [126] aim to systematize the different collaborative RA works that followed SEDA. The authors present the first step towards formalization of collabora-

tive RA, by providing a model for analyzing RA requirements (efficiency, soundness, and security). They design a collaborative RA protocol, called SAP (Synchronous Attestation Protocol), that adheres to the proposed model. SAP requires a swarm-wide protected synchronized clock on all devices. Unlike SEDA, SAP assumes a static swarm topology with a pre-established attestation tree, i. e., parent-child relation between all devices of the swarm are fixed. Unlike existing collaborative attestation techniques, DIAT allows efficient *control-flow* attestation in autonomous collaborative systems, enabling every collaborating devices to verify the integrity of all exchanged data.

6.4.2 *Integrity Enforcement*

An orthogonal approach to attestation is the enforcement of a device software's integrity on the device itself. Similar to attestation, the properties for which integrity is enforce, can be different, ranging from load-time integrity of code and static data in case of secure boot, enforcing programs' correct control-flow with Control-Flow Integrity (CFI), or the enforcement of data integrity with Data-Flow Integrity (DFI).

6.4.2.1 *Secure Boot*

With secure boot, integrity of a device's configuration is verified by the device itself rather than by an external entity [23] (cf. Section 2.2.1). Secure boot ensures that only a known and trustworthy software can be loaded on a device. The RoT is a small bootloader, which computes a hash of the content loaded into memory, and compares this to a signed hash stored in secure ROM. Hence, secure boot is limited to verifying software integrity of a device at load time. Attestation, in contrast, enables integrity verification of a system at any point in time.

6.4.2.2 *Control-flow Integrity*

CFI is a defense mechanism against run-time attacks. Modern run-time attacks do not inject or modify the code of a system. Instead, they reuse the existing code by hijacking the control flow of a program in order to cause unintended, malicious program behavior [328]. These attacks have been demonstrated on various platforms and devices, including embedded architectures such as ARM [232], SPARC [76] and Atmel AVR [146].

CFI's goal is to prevent that a program's control flow deviates from the developer-intended control flow. The integrity of the program flow is ensured by validating for each control-flow decision if the executed path lies within the program's Control-Flow Graph (CFG) [1, 2]. In particular, for each indirect branch instruction the branch target is checked against a set of valid targets and the branch is only executed if the check passed, otherwise the program is stopped.

CFI requires that the code of the program is modified, by instrumenting it with additional control-flow checks on each indirect branch. Furthermore, CFI relies on the assumption that the code on the device is immutable, i. e., an adversary cannot manipulate the CFI checks. This is usually achieved by assuming that the binary from which the program is loaded is untampered and that the code section in memory is write-protected.

6.4.2.3 *Data Integrity*

Data-Flow Integrity (DFI) [89] is a concept to prevent run-time attacks by preserving the integrity of data through additional run-time checks on data operations, i. e., DFI ensures that a program's data-flow complies with the program's Data-Flow Graph (DFG).

Noorman et al. [288] propose a system architecture and programming model that enforces data-flow between protected modules on distributed embedded devices. Their approach uses attestation to provision secret data to modules. These secrets are later used to authenticate the interaction of the modules. Unlike DIAT, they do not consider run-time attacks on the software modules. Also, their approach statically defines legitimate data-flows at compile-time, while DIAT dynamically tracks data-flows.

Memory safety is another approach that can ensure the integrity of a program's data at run time, e. g., by enforcing memory safety in programming languages [214, 283], by exploiting dynamic tainting [107, 97] or applying bound checking [332, 216].

Unfortunately, these solutions have large overhead, rely on special languages, struggle with high false positives rate, or are incapable of detecting all types of run-time attacks. More importantly, such enforcement techniques are not applicable to safety-critical real-time systems. DIAT provides integrity guarantees for data generated by a device's software by augmenting data with relevant control-flow information about their generation. It enables the detection of a large spectrum of run-time attacks. DIAT achieves this with low overhead for both, prover and verifier devices. Because DIAT follows the attestation paradigm integrity violations can be handled more prudent, allowing its application in safety critical systems.

7 DISCUSSION AND CONCLUSION

Connected systems are on the rise due to their promises to save cost while providing additional and improved functionalities. However, their connectivity opens these systems up to a wide variety of attacks, primarily remote attack via network.

To counter these threats, industry and academia have developed various security solutions. These solutions follow different strategies, for instance preventing attacks by isolating security critical system components from access by untrusted entities. In particular, Trusted Execution Environments (TEEs) allow the separation of a computing device's code and data into isolated entities that are protected from each other. However, providing comprehensive isolation is a challenging task, as has been demonstrated, for instance, by many side-channel attacks that emerged after the release of Intel TEE solution, called Software Guard Extensions (SGX). Other strategies aim to detect attacked and compromised devices. In particular, Remote Attestation (RA) is a security service that enables a remote verifier to validate the state of a potential compromised prover device.

The main objectives of this dissertation are the advancement of TEE functionalities, by developing new security architectures as well as extending and enhancing existing TEE solutions, and the advancement of RA schemes, by making them applicable to connected large-scale systems.

Subsequently, we briefly summarize the main results of this dissertation in Section 7.1 and discuss future research directions in Section 7.2.

7.1 Summary of Dissertation

In Chapter 3, we introduce two novel security architectures. TyTAN provides Trusted Execution Environments (TEEs) for low-end embedded systems while ensuring that the system's real-time guarantees are not violated by the TEE's security mechanism [62]. It enables isolated execution of mutually distrusting tasks, secure communication between tasks, and provides security services such as Remote Attestation (RA) and secure storage. Sanctuary enables strongly isolated use-space enclaves, i. e., TEE instances, on TrustZone-enabled systems [73]. Unlike TrustZone, Sanctuary can tolerate potentially malicious enclaves by isolating each enclave instance on a temporarily exempted CPU core and using strong hardware-based physical memory partitioning.

In Chapter 4, we investigate information leakage of TEEs. We focus on Intel's Software Guard Extensions (SGX) and develop a novel cache side-channel attack that can extract sensitive information from SGX enclaves [68]. We leverage the adversary's amplified capabilities, i. e., control over all system resources, in the context of SGX, and show that highly sensitive information can be extracted from enclaves due to their lack of side-

channel protection. With HardIDX, we present an efficient SGX-protected database index that allows secure outsourcing of databases, e. g., to untrusted cloud providers [149, 151]. HardIDX prevents information leakage through observable accesses patterns to the encrypted database stored in untrusted memory that occur while processing search queries. DR.SGX is an automated side-channel defense mechanism for SGX enclaves [72]. It uses our novel concept, called semantic-agnostic randomization, to obfuscate the location of all data in enclave memory, and thus, rendering an adversary's observations of memory access patterns through any side channel worthless.

In Chapter 5, we present two TEE-based applications. VoiceGuard uses SGX to enable privacy-preserving speech recognition [71]. While our mechanism to remotely enforce usage policies on smart devices, such as smartphones, utilizes ARM TrustZone [64]. Device usage policies can be enforced through selective control of smart device's peripherals such as camera or cellular network modem, while asserting the device owner's security and privacy.

In Chapter 6, we advance RA to become applicable to new scenarios and use cases that go beyond the typical setting, i. e., settings with a single prover device and a trusted verifier, that most attestation schemes assume. We show how to prevent malicious entities from misusing RA protocols to launch Denial-of-Service (DoS) attacks on prover devices [65]. With SEDA, we present the first swarm attestation protocol, which allows the efficient attestation of large groups of devices [34]. While DIAT presents a new paradigm in RA. Instead of attesting the integrity of devices, DIAT focuses on the integrity of data exchanged between collaborating devices [4]. This novel approach enables the efficient use of sophisticated attestation methods, such as run-time attestation, in setting were resource-constrained devices need to act as both, prover and verifier.

7.2 Future Research Directions

Trusted Execution Environments (TEEs) promise the strengthening system's security. The compromise of most parts of the system should not endanger the security of the TEE. However, various incidents, for instance the side-channel attack against Software Guard Extensions (SGX) enclaves presented in this work [68] and by others [412, 389, 340, 277, 174, 176], or the Foreshadow attack [390], have proven that it is challenging to build comprehensive security solutions that are completely immune to attacks.

Future systems, such as connected cars, will have much longer lifetimes than typical computing devices today. For these long-living systems, we need sustainable security solutions that can preserve their security guarantees throughout the systems' entire lifetime that can easily exceed 15 years. This calls for a new approach to system security because it is unrealistic to attempt to build a system that will be completely immune to all attacks for 15 year or longer. Such systems would need to resist all existing attacks as well as all new attack techniques developed in the future for as long as the system is in use. Thus, rather than trying to prevent system compromise we need to design systems that can tolerate compromise. For this, we need to make sure that a compromised system can be detected reliably by others, to prevent it from having adverse impact

on other connected devices. Additionally, we need methods to handle compromised devices. In particular, we need methods to recover and heal compromised systems. We need solutions that allow the benign owner to regain control over a system after an adversary has compromised a system and gained control over the system. However, just regaining control is not sufficient as the adversary could attack the system again, hence, we additionally need methods for systems to be able to self-adapt to previously unknown threats.

8 ABOUT THE AUTHOR

Ferdinand Brasser is research assistant at the Technische Universität Darmstadt and the Intel Collaborative Research Institute for Collaborative Autonomous Resilient Systems (ICRI-CARS), Germany. In 2013, he received his M.Sc. in IT-Security from TU Darmstadt, Germany. His research has mainly focused on Trusted Execution Environment (TEE) technologies and Remote Attestation (RA) protocols. He advanced RA to be applicable in large connected systems. Furthermore, he developed new TEE architectures for mobile and embedded systems. He researched side-channel leakage in TEEs by developing side-channel attacks as well as defenses, and showed applications of TEEs.

Awards

- Finalist Applied Research Challenge CSAW Europe 2017
- Best Paper Award for "HardIDX: Practical and Secure Index with SGX" at DBSec 2016 [149]

Academic Activities

- Student/Shadow Program Committee Member for ACM ASIA Conference on Computer and Communications Security (AsiaCCS) 2017
- Program Committee Member for ACM Workshop on Cyber-Physical Systems Security & Privacy (CPS-SPC) 2019
- Program Committee Member for 4th Workshop on System Software for Trusted Execution (SysTEX) 2019
- Reviewer for Security & Privacy Magazine – Special Issue: Hardware-Assisted Security 2019

Peer-reviewed Publications

1. Franz Ferdinand Brasser, Sven Bugiel, Atanas Filyanov, Ahmad-Reza Sadeghi, and Steffen Schulz. Softer Smartcards: Usable Cryptographic Tokens with Secure Execution. In *Proceedings of Financial Cryptography and Data Security* (FC), 2012. [74]

2. F. Ferdinand Brasser, Mihai Bucicoiu, and Ahmad-Reza Sadeghi. Swap and Play: Live Updating Hypervisors and Its Application to Xen. In *Proceedings of ACM Workshop on Cloud Computing Security* (CCSW), 2014. [61]

3. Ferdinand Brasser, Patrick Koeberl, Brahim El Mahjoub, Ahmad-Reza Sadeghi, and Christian Wachsmann. TyTAN: Tiny Trust Anchor for Tiny Devices. In *Proceedings of IEEE/ACM Design Automation Conference* (DAC), 2015. [62]

4. N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. SEDA: Scalable Embedded Device Attestation. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security* (CCS), 2015. [34]

5. Ferdinand Brasser, Kasper Rasmussen, Ahmad-Reza Sadeghi, and Gene Tsudik. Remote Attestation for Low-End Embedded Devices: the Prover's Perspective. In *Proceedings of IEEE/ACM Design Automation Conference* (DAC), 2016. [65]

6. Ferdinand Brasser, Vinod Ganapathy, Liviu Iftode, Daeyoung Kim, Christopher Liebchen, and Ahmad-Reza Sadeghi. Regulating ARM TrustZone Devices in Restricted Spaces. In *Proceedings of International Conference on Mobile Systems, Applications, and Services* (MobiSys), 2016. [64]

7. Luis Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *Proceedings of Annual Network and Distributed System Security Symposium* (NDSS), 2017. [152]

8. Bernardo Portela, Manuel Barbosa, Guillaume Scerri, Bogdan Warinschi, Raad Bahmani, Ferdinand Brasser, and Ahmad-Reza Sadeghi. Secure Multiparty Computation from SGX. In *Proceedings of Financial Cryptography and Data Security* (FC), 2017. [315]

9. Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and Secure Index with SGX. In *Proceedings of Conference on Data and Applications Security and Privacy* (DBSec), 2017. [149]

10. Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *Proceedings of USENIX Security Symposium* (USENIX Security), 2017. [67]

11. Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of USENIX Workshop on Offensive Technologies* (WOOT), 2017. [68]

12. Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and Secure Index with SGX in a Malicious Environmen. *Journal of Computer Security*, 26(5), 2018. [151]

13. Hamid Reza Ghaeini, Daniele Antonioli, Ferdinand Brasser, Ahmad-Reza Sadeghi, and Nils Ole Tippenhauer. State-Aware Anomaly Detection for Industrial Control Systems. In *Proceedings of ACM SIGAPP Symposium on Applied Computing* (SAC), 2018. [159]

14. Ferdinand Brasser, Tommaso Frassetto, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, and Christian Weinert. VoiceGuard: Secure and Private Speech Processing. In *Proceedings of Interspeech*, 2018. [71]

15. Andreas Schaad, Bjoern Grohmann, Oliver Winzenried, Ferdinand Brasser, and Ahmad-Reza Sadeghi. Towards a Cloud-based System for Software Protection and Licensing. In *Proceedings of International Joint Conference on e-Business and Telecommunications* (SECRYPT), 2018. [338]

16. Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *Proceedings of Annual Network and Distributed System Security Symposium* (NDSS), 2019. [4]

17. Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with User-space Enclaves. In *Proceedings of Annual Network and Distributed System Security Symposium* (NDSS), 2019. [73]

18. Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *Proceedings of Annual Network and Distributed System Security Symposium* (NDSS), 2019. [406]

19. Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. RIP-RH: Preventing Rowhammer-based Inter-Process Attacks. In *Proceedings of ACM ASIA Conference on Computer and Communications Security* (ASIA-CCS), 2019. [53]

20. Andrew Paverd, Marcus Völp, Ferdinand Brasser, Matthias Schunter, N. Asokan, Ahmad-Reza Sadeghi, Paulo Esteves-Veríssimo, Andreas Steininger, and Thorsten Holz. Sustainable Security & Safety: Challenges and Opportunities. In *Proceedings of 4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems* (CERTS), 2019. [304]

21. Hamid Reza Ghaeini, Matthew Chan, Raad Bahmani, Ferdinand Brasser, Luis Garcia, Jianying Zhou, Ahmad-Reza Sadeghi, Nils Ole Tippenhauer, and Saman Zonouz. PAtt: Physics-based Attestation of Control Systems. In *Proceedings of 22nd International Symposium on Research in Attacks, Intrusions and Defenses* (RAID), 2019. [160]

22. Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable

Side-Channel Protection for SGX using Data Location Randomization. In *Proceedings of Annual Computer Security Applications Conference* (ACSAC), 2019. [72]

23. Sridhar Adepu, Ferdinand Brasser, Luis Garcia, Michael Rodler, Lucas Davi, Ahmad-Reza Sadeghi, and Saman Zonouz. Control Behavior Integrity for Distributed Cyber-Physical Systems. In *Proceedings of IEEE/ACM Conference on Cyber-Physical Systems* (ICCPS), 2020. [8]

Special Session Papers

1. Ferdinand Brasser, Lucas Davi, Abhijitt Dhavlle, Tommaso Frassetto, Sai Manoj Pudukotai Dinakarrao, Setareh Rafatirad, Ahmad-Reza Sadeghi, Avesta Sasan, Hossein Sayadi, Shaza Zeitouni, and Houman Homayoun. Special Session: Advances and Throwbacks in Hardware-assisted Security. In *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (CASES), 2018. [70]

Technical Reports

1. Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. *arXiv:1611.08396v2*, 2016. [63]

2. Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. *arXiv:1702.07521*, 2017. [69]

3. Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and Secure Index with SGX. *arXiv:1703.04583*, 2017. [150]

4. Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *arXiv:1709.09917*, 2017. [66]

5. Sridhar Adepu, Ferdinand Brasser, Luis Garcia, Michael Rodler, Lucas Davi, Ahmad-Reza Sadeghi, and Saman Zonouz. Control Behavior Integrity for Distributed Cyber-Physical Systems. *arXiv:1812.08310*, 2018. [7]

6. Andrew Paverd, Marcus Völp, Ferdinand Brasser, Matthias Schunter, N. Asokan, Ahmad-Reza Sadeghi, Paulo Esteves-Veríssimo, Andreas Steininger, and Thorsten Holz. Sustainable Security & Safety: Challenges and Opportunities. `http://www.icri-cars.org/icri-cars/vision/`, 2018. [303]

7. Tigist Abera, Ferdinand Brasser, Lachlan J. Gunn, David Koisser, and Ahmad-Reza Sadeghi. SADAN: Scalable Adversary Detection in Autonomous Networks. *arXiv:1910.051903*, 2019. [5]

BIBLIOGRAPHY

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security*, 13(1), 2009.

[3] 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)*, 2016.

[4] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[5] Tigist Abera, Ferdinand Brasser, Lachlan J. Gunn, David Koisser, and Ahmad-Reza Sadeghi. SADAN: Scalable Adversary Detection in Autonomous Networks. https://arxiv.org/abs/1910.05190, 2019.

[6] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New Results on Instruction Cache Attacks. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2010.

[7] Sridhar Adepu, Ferdinand Brasser, Luis Garcia, Michael Rodler, Lucas Davi, Ahmad-Reza Sadeghi, and Saman Zonouz. Control Behavior Integrity for Distributed Cyber-Physical Systems. https://arxiv.org/abs/1812.08310, 2018.

[8] Sridhar Adepu, Ferdinand Brasser, Luis Garcia, Michael Rodler, Lucas Davi, Ahmad-Reza Sadeghi, and Saman Zonouz. Control Behavior Integrity for Distributed Cyber-Physical Systems. In *IEEE/ACM Conference on Cyber-Physical Systems (ICCPS)*, 2020.

[9] Advanced Micro Devices (AMD). AMD64 Architecture Programmer's Manual. http://developer.amd.com/wordpress/media/2012/10/24593_APM_v2.pdf, 2012.

[10] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order Preserving Encryption for Numeric Data. In *International Conference on Management of Data (SIGMOD)*, 2004.

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

[12] Al Danial. cloc – Count Lines of Code. https://github.com/AlDanial/cloc, 2020.

[13] Turky N. Al-Otaiby, Mohsen AlSherif, and Walter P. Bond. Toward Software Requirements Modularization Using Hierarchical Clustering Techniques. In *Annual Southeast Regional Conference (ACM SE)*, 2005.

[14] Eduardo Almentero, Julio Cesar Sampaio do Prado, and Carlos Lucena. Towards Software Modularization from Requirements. In *ACM SIGAPP Symposium On Applied Computing (SAC)*, 2014.

[15] Amazon Web Services, Inc. FreeRTOS – Real-time operating system for microcontrollers. https://www.freertos.org/, 2020.

[16] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. SANA: Secure and Scalable Aggregate Network Attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[17] Mahmoud Ammar, Mahdi Washha, and Bruno Crispo. WISE: Lightweight Intelligent Swarm Attestation Scheme for IoT (The Verifier's Perspective). In *International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, 2018.

[18] Mahmoud Ammar, Mahdi Washha, Gowri Ramachandran, and Bruno Crispo. SlimIoT: Scalable Lightweight Attestation Protocol for the Internet of Things. In *IEEE Conference on Dependable and Secure Computing (DSC)*, 2018.

[19] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. SμV–The Security MicroVisor: A Formally-Verified Software-Based Security Architecture for the Internet of Things. *IEEE Transactions on Dependable and Secure Computing*, 16(5), 2019.

[20] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[21] Nick Anderson and Valerie Strauss. Cheating concerns force delay in SAT scores for South Koreans and Chinese. https://www.washingtonpost.com/local/education/cheating-concerns-force-delay-in-sat-scores-for-south-koreans-and-chinese/2014/10/30/8e64bd9e-6056-11e4-9f3a-7e28799e0549_story.html, October 2014.

[22] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[23] William A. Arbaugh, David J. Farbert, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 1997.

[24] ARM Limited. ARM Security Technology: Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2008.

[25] 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.

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

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

[28] ARM Limited. GlobalPlatform based Trusted Execution Environment and TrustZone Ready. https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-00-51-36/GlobalPlatform-based-Trusted-Execution-Environment-and-TrustZone-R.pdf, 2013.

[29] ARM Limited. ARM Architecture Reference Manual. http://silver.arm.com/download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet9/DDI0487A_h_armv8_arm.pdf, 2015.

[30] ARM Limited. TrustZone Technology for ARMv8-M Architecture. https://static.docs.arm.com/100690/0200/armv8m_trustzone_technology_100690_0200.pdf, 2017.

[31] ARM Limited. Isolation using virtualization in the Secure world. https://developer.arm.com/-/media/Files/pdf/Isolation_using_virtualization_in_the_Secure_World_Whitepaper.pdf, 2018.

[32] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A Security Framework for the Analysis and Design of Software Attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

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

[34] N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. SEDA: Scalable Embedded Device Attestation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[35] Ahmed Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[36] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[37] Sumit Bajaj and Radu Sion. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 26(3), 2014.

[38] Arati Baliga, Pandurang Kamat, and Liviu Iftode. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2007.

[39] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting Kernel-Level Rootkits Using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5), September 2011.

[40] Elaine Barker and John Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. https://doi.org/10.6028/NIST.SP.800-90Ar1, 2015.

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

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

[43] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. http://eprint.iacr.org/2013/404.pdf.

[44] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and Efficiently Searchable Encryption. In *International Cryptology Conference (CRYPTO)*, 2007.

[45] Mihir Bellare, Phillip Rogaway, and Terence Spies. The FFX Mode of Operation for Format-Preserving Encryption. http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-spec.pdf, 2010.

[46] Gal Beniamini. Full TrustZone exploit for MSM8974. http://bits-please.blogspot.de/2015/08/full-trustzone-exploit-for-msm8974.html, 2015.

[47] Gal Beniamini. QSEE privilege escalation vulnerability and exploit. http://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, 2016.

[48] Gal Beniamini. Lifting the (Hyper) Visor: Bypassing Samsung's Real-Time Kernel Protection. https://googleprojectzero.blogspot.de/2017/02/lifting-hyper-visor-bypassing-samsungs.html, 2017.

[49] Gal Beniamini. Trust Issues: Exploiting TrustZone TEEs. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017.

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

[51] Sarani Bhattacharya and Debdeep Mukhopadhyay. Who watches the watchmen?: Utilizing Performance Monitors for Compromising keys of RSA on Intel Platforms. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2015.

[52] Blackberry. Enterprise Mobility Management - EMM. http://us.blackberry.com/enterprise/solutions/emm.html, 2016.

[53] Carsten Bock, Ferdinand Brasser, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. RIP-RH: Preventing Rowhammer-based Inter-Process Attacks. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2019.

[54] Andrey Bogdanov, Thomas Eisenbarth, Christof Paar, and Malte Wienecke. Differential Cache-Collision Timing Attacks on AES with Applications to Embedded CPUs. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2010.

[55] Aniruddha Bohra, Iulian Neamtiu, and Florin Sultan. Remote Repair of Operating System State Using Backdoors. In *International Conference on Autonomic Computing (ICAC)*, 2004.

[56] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-Preserving Symmetric Encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2009.

[57] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *International Cryptology Conference (CRYPTO)*, 2011.

[58] Dan Boneh and Brent Waters. Conjunctive, Subset, and Range Queries on Encrypted Data. In *Conference on Theory of Cryptography (TCC)*, 2007.

[59] Dan Boneh, Amit Sahai, and Brent Waters. Functional Encryption: Definitions and Challenges. In *Conference on Theory of Cryptography (TCC)*, 2011.

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

[61] F. Ferdinand Brasser, Mihai Bucicoiu, and Ahmad-Reza Sadeghi. Swap and Play: Live Updating Hypervisors and Its Application to Xen. In *ACM Workshop on Cloud Computing Security (CCSW)*, 2014.

[62] Ferdinand Brasser, Patrick Koeberl, Brahim El Mahjoub, Ahmad-Reza Sadeghi, and Christian Wachsmann. TyTAN: Tiny Trust Anchor for Tiny Devices. In *IEEE/ACM Design Automation Conference (DAC)*, 2015.

[63] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks. https://arxiv.org/abs/1611.08396, 2016.

[64] Ferdinand Brasser, Vinod Ganapathy, Liviu Iftode, Daeyoung Kim, Christopher Liebchen, and Ahmad-Reza Sadeghi. Regulating ARM TrustZone Devices in Restricted Spaces. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[65] Ferdinand Brasser, Kasper Rasmussen, Ahmad-Reza Sadeghi, and Gene Tsudik. Remote Attestation for Low-End Embedded Devices: the Prover's Perspective. In *IEEE/ACM Design Automation Conference (DAC)*, 2016.

[66] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. https://arxiv.org/abs/1709.09917, 2017.

[67] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *USENIX Security Symposium*, 2017.

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

[69] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. https://arxiv.org/abs/1702.07521, 2017.

[70] Ferdinand Brasser, Lucas Davi, Abhijitt Dhavlle, Tommaso Frassetto, Sai Manoj Pudukotai Dinakarrao, Setareh Rafatirad, Ahmad-Reza Sadeghi, Avesta Sasan, Hossein Sayadi, Shaza Zeitouni, and Houman Homayoun. Special Session: Advances and Throwbacks in Hardware-assisted Security. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2018.

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

[72] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX using Data Location Randomization. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[73] 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.

[74] Franz Ferdinand Brasser, Sven Bugiel, Atanas Filyanov, Ahmad-Reza Sadeghi, and Steffen Schulz. Softer Smartcards: Usable Cryptographic Tokens with Secure Execution. In *Financial Cryptography and Data Security (FC)*, 2012.

[75] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating cache/timing attacks in AES and RSA software implementations. In *RSA Conference 2006, session DEV-203*, 2006.

[76] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2008.

[77] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *USENIX Security Symposium*, 2013.

[78] John Butler. *Fundamentals of Forensic DNA Typing*. Academic Press, 2009.

[79] BYTE Magazine and Uwe F. Mayer. BYTEmark benchmark (nbench), port to linux, 1995-2011. Original address http://www.tux.org/~mayer/linux/bmark.html, now archived at https://web.archive.org/web/20151215162836/http://www.tux.org/~mayer/linux/bmark.html.

[80] Seyit A. Camtepe and Bülent Yener. Key Distribution Mechanisms for Wireless Sensor Networks: a Survey. Technical report, Rensselaer Polytechnic Institute, 2005.

[81] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[82] Xavier Carpent, Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Lightweight Swarm Attestation: A Tale of Two LISA-s. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2017.

[83] Xavier Carpent, Karim ElDefrawy, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. Reconciling Remote Attestation and Safety-critical Operation on Simple IoT Devices. In *IEEE/ACM Design Automation Conference (DAC)*, 2018.

[84] Xavier Carpent, Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Temporal Consistency of Integrity-Ensuring Computations and Applications to Embedded Systems Security. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2018.

[85] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. ERASMUS: Efficient Remote Attestation via Self-Measurement for Unattended Settings. In *Conference on Design, Automation & Test in Europe (DATE)*, 2018.

[86] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. Remote Attestation via Self-Measurement. *ACM Transactions on Design Automation of Electronic Systems*, 24(1), 2018.

[87] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *International Cryptology Conference (CRYPTO)*, 2013.

[88] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the Difficulty of Software-Based Attestation of Embedded Devices. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[89] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[90] David Champagne and Ruby B. Lee. Scalable Architectural Support for Trusted Software. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

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

[92] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable, Efficient, and Scalable Secure Two-Party Computation.

Cryptology ePrint Archive. Report 2017/1109, 2017. https://eprint.iacr.org/2017/1109.pdf.

[93] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *USENIX Security Symposium*, 2011.

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

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

[96] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2017.

[97] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2005.

[98] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data Attacks Are Realistic Threats. In *USENIX Security Symposium*, 2005.

[99] Ke Cheng, Yulong Shen, Yongzhi Wang, Liangmin Wang, Jianfeng Ma, Xionghong Jiang, and Cuicui Su. Strongly Secure and Efficient Range Queries in Cloud Databases under Multiple Keys. In *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2019.

[100] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real Time Detection of Cache-based Side-channel Attacks Using Hardware Performance Counters. *Applied Soft Computing*, 49, 2016.

[101] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G. Edward Suh. Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking. In *Advances in Cryptology - ASIACRYPT*, 2003.

[102] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. Compiler Mitigations for Time Attacks on Modern x86 Processors. *ACM Transactions on Architecture and Code Optimization*, 8(4), 2012.

[103] John Clemens, Raj Pal, and Branden Sherrell. Runtime State Verification on Resource-Constrained Platforms. In *IEEE Military Communications Conference (MILCOM)*, 2018.

[104] Mauro Conti, Roberto Di Pietro, Luigi Vincenzo Mancini, and Alessandro Mei. Emergent Properties: Detection of the Node-capture Attack in Mobile Wireless Sensor Networks. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2008.

[105] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and William Tim Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, Internet Engineering Task Force (IETF), 2008. URL http://www.rfc-editor.org/rfc/rfc5280.txt.

[106] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2009.

[107] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain Internet worms? https://www.microsoft.com/en-us/research/publication/can-we-contain-internet-worms/, 2004.

[108] Miguel B. Costa, Nuno O. Duarte, Nuno Santos, and Paulo Ferreira. TrUbi: A System for Dynamically Constraining Mobile Devices Within Restrictive Usage Scenarios. In *International Symposium on Mobile Ad Hoc Networking and Computing (Mobihoc)*, 2017.

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

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

[111] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security Symposium*, 2014.

[112] Landon P. Cox and Peter M. Chen. Pocket Hypervisors: Opportunities and Challenges. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2007.

[113] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[114] Erol Şahin. Swarm Robotics: From Sources of Inspiration to Domains of Application. In *Swarm Robotics*, 2005.

[115] Ang Cui and Salvatore J. Stolfo. A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-area Scan. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.

[116] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. Tracking Rootkit Footprints with a Practical Memory Analysis System. In *USENIX Security Symposium*, 2012.

[117] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2006.

[118] Christoffer Dall and Jason Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[119] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[120] Wilfried Daniels, Danny Hughes, Mahmoud Ammar, Bruno Crispo, Nelson Matthys, and Wouter Joosen. SμV - the Security Microvisor: A Virtualisation-based Security Middleware for the Internet of Things. In *ACM/IFIP/USENIX Middleware Conference: Industrial Track (Middleware)*, 2017.

[121] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kaynar. A Logic of Secure Systems and its Application to Trusted Computing. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2009.

[122] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2013.

[123] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: Hardware-assisted Flow Integrity Extension. In *IEEE/ACM Design Automation Conference (DAC)*, 2015.

[124] Matt Day, Giles Turner, and Natalia Drozdiak. Amazon Workers Are Listening to What You Tell Alexa. https://www.bloomberg.com/news/articles/2019-04-10/is-anyone-listening-to-you-on-alexa-a-global-team-reviews-audio, 2019.

[125] Ruan de Clercq, Frank Piessens, Dries Schellekens, and Ingrid Verbauwhede. Secure Interrupts on Low-End Microcontrollers. In *Application-specific Systems, Architectures and Processors (ASAP)*, 2014.

[126] 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 *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[127] Ivan De Oliveira Nunes, Karim ElDefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *USENIX Security Symposium*, 2019.

[128] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. Practical Private Range Search Revisited. In *International Conference on Management of Data (SIGMOD)*, 2016.

[129] 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)*, 2017.

[130] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution. In *International Conference On Computer Aided Design (ICCAD)*, 2018.

[131] Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force (IETF), 2008. URL `http://www.rfc-editor.org/rfc/rfc5246.txt`.

[132] Daniel Dinu, Archanaa Santhana Krishnan, and Patrick Schaumont. SIA: Secure Intermittent Architecture for Off-the-Shelf Resource-Constrained Microcontrollers. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2019.

[133] Danny Dolev and Andrew C. Yao. On the Security of Public Key Protocols. In *Symposium on Foundations of Computer Science (SFCS)*, 1981.

[134] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4), 2012.

[135] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *International Conference on Machine Learning (ICML)*, 2016.

[136] F. Betül Durak, Thomas M. DuBuisson, and David Cash. What Else is Revealed by Order-Revealing Encryption? In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[137] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. The Untapped Potential of Trusted Execution Environments on Mobile Devices. *IEEE Security & Privacy*, 12(4), 2014.

[138] Karim ElDefrawy and Gene Tsudik. Opinion: Advancing Remote Attestation via Computer-aided Formal Verification of Designs and Synthesis of Executables. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2019.

[139] Karim ElDefrawy, Aurelien Francillon, Daniele Perito, and Gene Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[140] Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. HYDRA: Hybrid Design for Remote Attestation (Using a Formally Verified Microkernel). In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[141] Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Fusing Hybrid Remote Attestation with a Formally Verified Microkernel: Lessons Learned. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2017.

[142] European Parliament, Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). http://data.europa.eu/eli/reg/2016/679/oj, 2016.

[143] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[144] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *European Symposium on Research in Computer Security (ESORICS)*, 2015.

[145] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan J. Parno. Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software. In *ACM Symposium on Operating System Principles (SOSP)*, 2017.

[146] Aurélien Francillon and Claude Castelluccia. Code Injection Attacks on Harvard-architecture Devices. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2008.

[147] Aurélien Francillon, Quan Nguyen, Kasper B. Rasmussen, and Gene Tsudik. A Minimalist Approach to Remote Attestation. In *Conference on Design, Automation & Test in Europe (DATE)*, 2014.

[148] Jessie Frazelle. Open Source Firmware. *Communications of the ACM*, 62(10), 2019.

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

[150] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and Secure Index with SGX. https://arxiv.org/abs/1703.04583, 2017.

[151] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. HardIDX: Practical and Secure Index with SGX in a Malicious Environment. *Journal of Computer Security*, 26(5), 2018.

[152] Luis Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[153] Ryan W. Gardner, Sujata Garera, and Aviel D. Rubin. Detecting Code Alteration by Creating a Temporary Memory Bottleneck. *IEEE Transactions on Information Forensics and Security*, 2009.

[154] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In *International Cryptology Conference (CRYPTO)*, 2016.

[155] John Garofalo, David Graff, Doug Paul, and David S. Pallett. CSR-I,II (WSJo,1) Complete. Linguistic Data Consortium, Philadelphia, USA, 2007.

[156] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *Mobile Security Technologies (MOST)*, 2014.

[157] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *ACM Symposium on Theory of Computing (STOC)*, 2009.

[158] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic Evaluation of the AES Circuit. In *International Cryptology Conference (CRYPTO)*, 2012.

[159] Hamid Reza Ghaeini, Daniele Antonioli, Ferdinand Brasser, Ahmad-Reza Sadeghi, and Nils Ole Tippenhauer. State-Aware Anomaly Detection for Industrial Control Systems. In *ACM SIGAPP Symposium On Applied Computing (SAC)*, 2018.

[160] Hamid Reza Ghaeini, Matthew Chan, Raad Bahmani, Ferdinand Brasser, Luis Garcia, Jianying Zhou, Ahmad-Reza Sadeghi, Nils Ole Tippenhauer, and Saman Zonouz. PAtt: Physics-based Attestation of Control Systems. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[161] Girard, Olivier. openMSP430. http://opencores.org/project,openmsp430, 2009.

[162] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *USENIX Security Symposium*, 2012.

[163] Cornelius Glackin, Gérard Chollet, Nazim Dugan, Nigel Cannings, Julie Wall, Shahzaib Tahir, Indranil Ghosh Ray, and Muttukrishnan Rajarajan. Privacy Preserving Encrypted Phonetic Search of Speech Data. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017.

[164] Virgil Gligor. Establishing and Maintaining Root of Trust on Commodity Computer Systems. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2019.

[165] Virgil Gligor and Maverick S. L. Woo. Establishing Software Root of Trust Unconditionally. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2019.

[166] GlobalPlatform, Inc. TEE Management Framework (Version 1.0). https://www.globalplatform.org/specificationform.asp?fid=7866, 2016.

[167] GlobalPlatform, Inc. Introduction to Trusted Execution Environments. https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf, 2018.

[168] Oded Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[169] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 1996.

[170] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation. In *ACM Symposium on Discrete Algorithms (SODA)*, 2012.

[171] Google. Zircon. https://fuchsia.dev/fuchsia-src/zircon, 2018.

[172] Google. Android. https://www.android.com/, 2019.

[173] Johannes Götzfried, Tilo Müller, Ruan de Clercq, Pieter Maene, Felix C. Freiling, and Ingrid Verbauwhede. Soteria: Offline Software Protection within Low-cost Embedded Devices. In *Annual Computer Security Applications Conference (ACSAC)*, 2015.

[174] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *European Workshop on System Security (EuroSec)*, 2017.

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

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

[177] Alexey Gribov, Dhinakaran Vinayagamurthy, and Sergey Gorbunov. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proceedings on Privacy Enhancing Technologies*, 2019.

[178] William G. Griswold, Macneil Shonle, Kevin Sullivan, Yuanyuan Song, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23, 2006.

[179] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-Abuse Attacks against Order-Revealing Encryption. Cryptology ePrint Archive. Report 2016/895, 2016. https://eprint.iacr.org/2016/895.pdf.

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

[181] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2016.

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

[183] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution. In *Engineering Secure Software and Systems (ESSoS)*, 2018.

[184] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *USENIX Annual Technical Conference (ATC)*, 2017.

[185] Nadia Heninger and Hovav Shacham. Reconstructing RSA Private Keys from Random Key Bits. In *International Cryptology Conference (CRYPTO)*, 2009.

[186] Andri Puspo Heriyanto. Procedures And Tools For Acquisition And Analysis Of Volatile Memory On Android Smartphones. In *Australian Digital Forensics Conference (ADF)*, 2013.

[187] Alex Hern. Apple contractors 'regularly hear confidential details' on Siri recordings. https://www.theguardian.com/technology/2019/jul/26/apple-contractors-regularly-hear-confidential-details-on-siri-recordings, 2019.

[188] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. CryptoDL: Deep Neural Networks over Encrypted Data. https://arxiv.org/abs/1711.05189, 2017.

[189] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *USENIX Security Symposium*, 2014.

[190] Fiona Higgins, Allan Tomlinson, and Keith M. Martin. Threats to the Swarm: Security Considerations for Swarm Robotics. *International Journal on Advances in Security*, pages 288–297, 2009.

[191] Jason D. Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd My Gadgets Go? In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2012.

[192] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila Yavuz. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *Proceedings on Privacy Enhancing Technologies*, 2019.

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

[194] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring Operating System Kernel Integrity with OSck. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[195] George Hotz. Towelroot Android Root Exploit. https://towelroot.com/, 2014.

[196] 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 (IEEE S&P)*, 2016.

[197] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vTZ: Virtualizing ARM TrustZone. In *USENIX Security Symposium*, 2017.

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

[199] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. DARPA: Device Attestation Resilient against Physical Attacks. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2016.

[200] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Shaza Zeitouni. SeED: Secure Non-interactive Attestation for Embedded Devices. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[201] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. US-AID: Unattended Scalable Attestation of IoT Devices. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[202] Alberto Garcia Illera and Javier Vazquez Vidal. Lights Off! The Darkness of the Smart Meters. In *Blackhat Europe*, 2014.

[203] Intel Corporation. An Overview of Cache. https://web.archive.org/web/20070211091515/http://download.intel.com/design/intarch/papers/cache6.pdf, 1997.

[204] Intel Corporation. Intel Trusted Execution Technology (Intel TXT) – Software Development Guide. https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf, 2009.

[205] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2012.

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

[207] Intel Corporation. Intel(R) Software Guard Extensions for Linux* OS. https://github.com/intel/linux-sgx/, 2017.

[208] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. http://www-ssl.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 2017.

[209] Intel Corporation. Deep Dive: Intel Analysis of L1 Terminal Fault. https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-l1-terminal-fault, 2018.

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

[211] Pawan Jadia and Anish Mathuria. Efficient Secure Aggregation in Sensor Networks. In *Efficient Secure Aggregation in Sensor Networks (HiPC)*, 2004.

[212] Hassaan Janjua, Mahmoud Ammar, Bruno Crispo, and Danny Hughes. Towards a Standards-compliant Pure-software Trusted Execution Environment for Resource-constrained Embedded Devices. In *System Software for Trusted Execution (SysTEX)*, 2019.

[213] Joint Test Action Group (JTAG). 1149.1-2013 - IEEE Standard for Test Access Port and Boundary-scan Architecture. http://standards.ieee.org/findstds/standard/1149.1-2013.html, 2013.

[214] Richard W. M. Jones and Paul H. J. Kelly. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Automated and Algorithmic Debugging (AADEBUG)*, 1997.

[215] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security Symposium*, 2018.

[216] Samuel C. Kendall. Bcc: run-time checking for C programs. In *USENIX Summer Conference*, 1983.

[217] Rick Kennell and Leah H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *USENIX Security Symposium*, 2003.

[218] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N. Serpanos, and Stefanos Kaxiras. Non deterministic caches: a simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3), 2008.

[219] Florian Kerschbaum and Axel Schröpfer. Optimal Average-Complexity Ideal-Security Order-Preserving Encryption. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[220] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[221] Chongkyung Kil, Emre C. Sezer, Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2009.

[222] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. ShieldStore: Shielded In-memory Key-value Storage with SGX. In *European Conference on Computer Systems (EuroSys)*, 2019.

[223] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems*, 32(1), 2014.

[224] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *International Cryptology Conference (CRYPTO)*, 1999.

[225] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

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

[227] Patrick Koeberl, Vinay Phegade, Anand Rajan, Thomas Schneider, Steffen Schulz, and Maria Zhdanova. Time to Rethink: Trust Brokerage Using Trusted Execution Environments. In *International Conference on Trust and Trustworthy Computing (TRUST)*, 2015.

[228] Florian Kohnhäuser and Stefan Katzenbeisser. Secure Code Updates for Mesh Networked Commodity Low-End Embedded Devices. In *European Symposium on Research in Computer Security (ESORICS)*, 2016.

[229] Florian Kohnhäuser, Niklas Büscher, Sebastian Gabmeyer, and Stefan Katzenbeisser. SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[230] Joonho Kong, Farinaz Koushanfar, Praveen K. Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. PUFatt: Embedded Platform Attestation Based on Novel Processor-Based PUFs. In *IEEE/ACM Design Automation Conference (DAC)*, 2014.

[231] Robert Könighofer. A Fast and Cache-Timing Resistant Implementation of the AES. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2008.

[232] Tim Kornau. Return Oriented Programming for the ARM Architecture. https://static.googleusercontent.com/media/www.zynamics.com/de//downloads/kornau-tim--diplomarbeit--rop.pdf, 2009.

[233] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental Security Analysis of a Modern Automobile. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2010.

[234] Kari Kostiainen, Jan-Erik Ekberg, N. Asokan, and Aarne Rantala. On-board Credentials with Open Provisioning. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2009.

[235] Nikos Koutroumpouchos, Christoforos Ntantogian, Sofia Menesidou, Kaitai Liang, Panagiotis Gouvas, Christos Xenakis, and Thanassis Giannetsos. Secure Edge Computing with Lightweight Control-Flow Property-based Attestation. In *IEEE Conference on Network Softwarization (NetSoft)*, 2019.

[236] Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, and John Butterworth. New Results for Timing-Based Attestation. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2011.

[237] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Internet Engineering Task Force (IETF), February 1997. URL https://tools.ietf.org/html/rfc2104.

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

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

[240] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. SeCloak: ARM Trustzone-based Mobile Peripheral Control. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[241] Matej Lexa and Giorgio Valle. PRIMEX: Rapid identification of oligonucleotide matches in whole genomes. *Bioinformatics*, 2003. https://www.researchgate.net/publication/233734306_mex-099tar.

[242] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.

[243] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. SBAP: Software-Based Attestation for Peripherals. In *International Conference on Trust and Trustworthy Computing (TRUST)*, 2010.

[244] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: Verifying the Integrity of PERipherals Firmware. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2011.

[245] Linaro. Open Portable Trusted Execution Environment. https://www.op-tee.org/, 2018.

[246] Linaro. OP-TEE documentation – Pager. https://optee.readthedocs.io/en/latest/architecture/core.html#pager, 2019.

[247] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[248] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. Hypervisor Support for Identifying Covertly Executing Binaries. In *USENIX Security Symposium*, 2008.

[249] Chang Liu, Michael Hicks, and Elaine Shi. Memory Trace Oblivious Program Execution. In *IEEE Computer Security Foundations Symposium (CSF)*, 2013.

[250] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

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

[252] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2015.

[253] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. Software Abstractions for Trusted Sensors. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[254] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN transformations. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[255] Jing Liu, Yang Xiao, Shuhui Li, Wei Liang, and C. L. Philip Chen. Cyber Security and Privacy Issues in Smart Grids. *IEEE Communications Surveys Tutorials*, pages 981–997, 2012.

[256] LLVM Foundation. The LLVM Compiler Infrastructure. https://llvm.org, 2019.

[257] Yanbin Lu. Privacy-preserving Logarithmic-time Search on Encrypted Data in Cloud. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[258] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[259] Pieter Maene, Johannes Götzfried, Ruan de Clercq, Tilo Müller, Felix C. Freiling, and Ingrid Verbauwhede. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Transactions on Computers*, 67(3), 2018.

[260] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[261] Ahmed Mahmoud, Ulrich Rührmair, Mehrdad Majzoobi, and Farinaz Koushanfar. Combined Modeling and Side Channel Attacks on Strong PUFs. Cryptology ePrint Archive. Report 2013/632, 2013. https://eprint.iacr.org/2013/632.pdf.

[262] Maja Malenko and Marcel Baunach. Hardware/Software Co-designed Peripheral Protection in Embedded Devices. In *IEEE International Conference on Industrial Cyber Physical Systems (ICPS)*, 2019.

[263] Avradip Mandal, John C. Mitchell, Hart Montgomery, and Arnab Roy. Data Oblivious Genome Variants Search on Intel SGX. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology (DPM)*, 2018.

[264] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *European Conference on Computer Systems (EuroSys)*, 2008.

[265] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

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

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

[268] Carlo Maria Medaglia and Alexandru Serbanati. An Overview of Privacy and Security Issues in the Internet of Things. In *The Internet of Things*, pages 389–395. Springer, 2010.

[269] Mary Meeker. 2015 Internet Trends. https://www.kleinerperkins.com/perspectives/2015-internet-trends, 2015.

[270] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. PX4: A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2015.

[271] Mellanox Technologies Inc. Introduction to InfiniBand. http://www.mellanox.com/blog/2014/09/introduction-to-infiniband, September 2014.

[272] Microsoft. Microsoft Intune documentation. https://docs.microsoft.com/en-us/intune/, 2019.

[273] Alex Migicovsky, Zakir Durumeric, Jeff Ringenberg, and J. Alex Halderman. Outsmarting Proctors with Smartwatches: A Case Study on Wearable Computing Security. In *Financial Cryptography and Data Security (FC)*, 2014.

[274] Charlie Miller and Christopher Valasek. A Survey of Remote Automotive Attack Surfaces. In *Blackhat USA*, 2014.

[275] Saeed Mirzamohammadi, Justin A. Chen, Ardalan Amiri Sani, Sharad Mehrotra, and Gene Tsudik. Ditio: Trustworthy Auditing of Sensor Activities in Mobile & IoT Devices. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2017.

[276] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Popa. Oblix: An Efficient Oblivious Search Index. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2018.

[277] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies The Power of Cache Attacks. In *Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[278] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations in SGX. In *The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2018.

[279] Payman Mohassel and Yupeng Zhang. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2017.

[280] National Institute of Standards and Technology. Digital Signature Standard (DSS). https://doi.org/10.6028/NIST.FIPS.186-5-draft, 2019.

[281] Muhammad Naveed. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Cryptology ePrint Archive. Report 2015/668, 2015. https://eprint.iacr.org/2015/668.pdf.

[282] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference Attacks on Property-Preserving Encrypted Databases. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[283] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), 2005.

[284] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A Refined Look at Bernstein's AES Side-Channel Analysis. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2006.

[285] Zhenyu Ning and Fengwei Zhang. Ninja: Towards Transparent Tracing and Debugging on ARM. In *USENIX Security Symposium*, 2017.

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

[287] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Transactions on Privacy and Security*, 20(3), 2017.

[288] Job Noorman, Jan Tobias Mühlberg, and Frank Piessens. Authentic Execution of Distributed Event-Driven Applications with a Small TCB. In *International Workshop on Security and Trust Management (STM)*, 2017.

[289] NXP Semiconductors. AN4581 – i.MX Secure Boot on HABv4 Supported Devices. https://www.nxp.com/docs/en/application-note/AN4581.pdf, 2020.

[290] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and Preventing Leakage in MapReduce. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[291] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aasthaa Meht, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*, 2016.

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

[293] Oracle. Java Card – The Open Application Platform for Secure Elements. https://www.oracle.com/technetwork/java/javacard/overview/java-card-data-sheet-19-01-07-5250140.pdf, 2019.

[294] 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.

[295] Emmanuel Owusu, Jorge Guajardo, Jonathan M. McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[296] Daniel Page. Defending Against Cache Based Side-Channel Attacks. *Information Security Technical Report*, 8(1), 2003.

[297] Daniel Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. Cryptology ePrint Archive. Report 2005/280, 2005. http://eprint.iacr.org/2005/280.pdf.

[298] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2012.

[299] Haemin Park, Dongwon Seo, Heejo Lee, and Adrian Perrig. SMATT: Smart Meter Attestation Using Multiple Target Selection and Copy-Proof Memory. In *Computer Science and its Applications*, 2012.

[300] Bryan J. Parno. Bootstrapping Trust in a "Trusted" Platform. In *Workshop on Hot Topics in Security (HotSec)*, 2008.

[301] Bryan J. Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping Trust in Commodity Computers. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2010.

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

[303] Andrew Paverd, Marcus Völp, Ferdinand Brasser, Matthias Schunter, N. Asokan, Ahmad-Reza Sadeghi, Paulo Esteves-Veríssimo, Andreas Steininger, and Thorsten Holz. Sustainable Security & Safety: Challenges and Opportunities. https://www.icri-cars.org/icri-cars/vision/, 2018.

[304] Andrew Paverd, Marcus Völp, Ferdinand Brasser, Matthias Schunter, N. Asokan, Ahmad-Reza Sadeghi, Paulo Esteves-Veríssimo, Andreas Steininger, and Thorsten Holz. Sustainable Security & Safety: Challenges and Opportunities. In *International Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS)*, 2019.

[305] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2001.

[306] Mathias Payer. HexPADS: A Platform to Detect "Stealth" Attacks. In *Engineering Secure Software and Systems (ESSoS)*, 2016.

[307] Siani Pearson, Marco Casassa Mont, and Stephen Crane. Persistent and Dynamic Trust: Analysis and the Related Impact of Trusted Platforms. In *Trust Management*, 2005.

[308] Colin Percival. Cache Missing for Fun and Profit. In *The Technical BSD Conference (BSDCan)*, 2005.

[309] Daniele Perito and Gene Tsudik. Secure Code Update for Embedded Devices via Proofs of Secure Erasure. In *European Symposium on Research in Computer Security (ESORICS)*, 2010.

[310] Nick L. Petroni and Michael Hicks. Automated Detection of Persistent Kernel Control-flow Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007.

[311] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*, 2004.

[312] Nick L. Petroni, Timothy Fraser, AAron Walters, and William A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *USENIX Security Symposium*, 2006.

[313] Jonathan Pollet and Joe Cummins. Electricity for Free? The Dirty Underbelly of SCADA and Smart Meters. In *Blackhat USA*, 2010.

[314] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing . In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[315] Bernardo Portela, Manuel Barbosa, Guillaume Scerri, Bogdan Warinschi, Raad Bahmani, Ferdinand Brasser, and Ahmad-Reza Sadeghi. Secure Multiparty Computation from SGX. In *Financial Cryptography and Data Security (FC)*, 2017.

[316] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. The Kaldi Speech Recognition Toolkit. In *Workshop on Automatic Speech Recognition and Understanding (ASRU)*, 2011.

[317] Patti Price, William M. Fisher, Jared Bernstein, and David S. Pallett. Resource Management RM1 2.0. Linguistic Data Consortium, Philadelphia, USA, 1993.

[318] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database Using SGX. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2018.

[319] Masoom Rabbani, Jo Vliegen, Jori Winderickx, Mauro Conti, and Nele Mentens. SHeLA: Scalable Heterogeneous Layered Attestation. *IEEE Internet of Things Journal*, 6(6), 2019.

[320] Himanshu Raj, Stefan Saroiu, Alec Wolman, and Jitu Padhye. Splitting the Bill for Mobile Data with SIMlets. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2013.

[321] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Firmware-based TPM 2.0 Implementation. Technical Report MSR-TR-2015-84, Microsoft Research, 2015.

[322] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems, 3rd Edition*. McGraw-Hill, 3rd edition, 2002.

[323] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-channels Through Obfuscated Execution. In *USENIX Security Symposium*, 2015.

[324] Justin Rattner. Extreme Scale Computing. Keynote at Annual International Symposium on Computer Architecture (ISCA), 2012.

[325] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants Count: Practical Improvements to Oblivious RAM. In *USENIX Security Symposium*, 2015.

[326] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2018.

[327] Ronald Linn Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21 (2), 1978.

[328] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1), 2012.

[329] Dan Rosenberg. Reflections on Trusting TrustZone. In *Blackhat USA*, 2014.

[330] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. DeepSecure: Scalable Provably-Secure Deep Learning. In *IEEE/ACM Design Automation Conference (DAC)*, 2018.

[331] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable Self-Assembly in a Thousand-Robot Swarm. *Science*, pages 378–380, 2014.

[332] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2004.

[333] Samsung. KNOX Workspace Supported MDMs. https://www.samsungknox.com/en/products/knox-workspace/technical/knox-mdm-feature-list, 2016.

[334] Samsung. Device-side Security: Samsung Pay, TrustZone, and the TEE. https://developer.samsung.com/tech-insights/pay/device-side-security, 2016.

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

[336] Vasily Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and R. Kapitza. STANlite – A Database Engine for Secure Data Processing at Rack-Scale Level. In *International Conference on Cloud Engineering (IC2E)*, 2018.

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

[338] Andreas Schaad, Bjoern Grohmann, Oliver Winzenried, Ferdinand Brasser, and Ahmad-Reza Sadeghi. Towards a Cloud-based System for Software Protection and Licensing. In *International Joint Conference on e-Business and Telecommunications (SECRYPT)*, 2018.

[339] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2015.

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

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

[342] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2004.

[343] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *ACM Symposium on Operating System Principles (SOSP)*, 2005.

[344] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SCUBA: Secure Code Update By Attestation in Sensor Networks. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2006.

[345] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

[346] Arvind Seshadri, Mark Luk, and Adrian Perrig. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *Distributed Computing in Sensor System (DCOSS)*, 2008.

[347] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007.

[348] Manali D. Shah, Shrenik N. Gala, and Narendra M. Shekokar. Lightweight Authentication Protocol used in Wireless Sensor Network. In *International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA)*, 2014.

[349] Umesh Shankar, Monica Chew, and J. D. Tygar. Side Effects Are Not Sufficient to Authenticate Software. In *USENIX Security Symposium*, 2004.

[350] Di Shen. Exploiting TrustZone on Android. In *Blackhat USA*, 2015.

[351] Di Shen. Defeating Samsung KNOX with zero privilege. In *Blackhat USA*, 2017.

[352] Emily Shen, Elaine Shi, and Brent Waters. Predicate Privacy in Encryption Systems. In *Conference on Theory of Cryptography (TCC)*, 2009.

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

[354] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2016.

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

[356] Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. A Compiler and Verifier for Page Access Oblivious Computation. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[357] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2013.

[358] Dawn Xiaoding Song, Davi Wagner, and Adria Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2000.

[359] Arnaud Soullie. Industrial Control Systems: Pentesting PLCs 101. In *Blackhat Europe*, 2014.

[360] Raphael Spreitzer and Benoît Gérard. Towards More Practical Time-Driven Cache Attacks. In *Information Security Theory and Practice. Securing the Internet of Things (WISTP)*, 2014.

[361] Raphael Spreitzer and Thomas Plos. Cache-Access Pattern Attack on Disaligned AES T-Tables. In *International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2013.

[362] John A. Stankovic and R. Rajkumar. Real-Time Operating Systems. *Real-Time Systems*, 28(2-3), 2004.

[363] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. https://arxiv.org/abs/1806.07480, 2018.

[364] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

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

[366] Andy Stevenson. Boot into Recovery Mode for Rooted and Un-rooted Android devices. http://androidflagship.com/605-enter-recovery-mode-rooted-un-rooted-android, 2014.

[367] Raoul Strackx and Frank Piessens. Ariadne: A Minimal Approach to State Continuity. In *USENIX Security Symposium*, 2016.

[368] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *Security and Privacy in Communication Networks (SecureComm)*, 2010.

[369] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *International Conference on Supercomputing (ICS)*, 2003.

[370] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. TrustDump: Reliable Memory Acquisition on Smartphones. In *European Symposium on Research in Computer Security (ESORICS)*, 2014.

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

[372] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. OAT: Attesting Operation Integrity of Embedded Devices. In *IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[373] Joe Sylve, Andrew Case, Lodovico Marziale, and Golden G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3), 2012.

[374] Benjamin Tan, Morteza Biglari-Abhari, and Zoran Salcic. An Automated Security-Aware Approach for Design of Embedded Systems on MPSoC. *ACM Transactions on Embedded Computing Systems*, 16(5s), 2017.

[375] Hailun Tan, Gene Tsudik, and Sanjay Jha. MTRA: Multiple-Tier Remote Attestation in IoT Networks. In *IEEE Conference on Communications and Network Security (CNS)*, 2017.

[376] Texas Instruments Incorporated. MSP Code Protection Features. https://www.ti.com/lit/an/slaa685/slaa685.pdf, 2015.

[377] The InfiniBand Trade Association. The InfiniBand Architecture Specification. https://www.infinibandta.org/, 2014.

[378] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. ScaRR: Scalable Runtime Remote Attestation for Complex Systems. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[379] Shruti Tople, Hung Dang, Prateek Saxena, and Ee-Chien Chang. PermuteRam: Optimizing Oblivious Computation for Efficiency. Cryptology ePrint Archive. Report 2017/885, 2015. https://eprint.iacr.org/2017/885.pdf.

[380] Amos Treiber, Andreas Nautsch, Jascha Kolberg, Thomas Schneider, and Christoph Busch. Privacy-Preserving PLDA Speaker Verification using Outsourced Secure Computation. *Speech Communication*, 114, 2019.

[381] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1), 2010.

[382] Trusted Computing Group (TCG). TPM Main Specification Level 2 Version 1.2. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, 2007.

[383] Trusted Computing Group (TCG). Trusted platform module. http://www.trustedcomputinggroup.org, 2011.

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

[385] Theodoros Tzouramanis. Secure Range Query Processing over Untrustworthy Cloud Services. In *International Database Engineering & Applications Symposium (IDEAS)*, 2017.

[386] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Virtual Machine Research And Technology Symposium (VM)*, 2004.

[387] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting Hardware Performance Counters. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2008.

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

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

[390] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[391] Abhishek Vijeev, Vinod Ganapathy, and Chiranjib Bhattacharyya. Regulating Drones in Restricted Spaces. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2019.

[392] Shivraj VL, Meena Thakur, and Rajan Ma. A Novel Multi Verifier Device Attestation Scheme for Swarm of Devices. In *International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, 2018.

[393] VMware. Verizon Wireless and VMware Securely Mix the Professional and Personal Mobile Experience with Dual Persona Android Devices, October 2011.

[394] Marcus Völp, Adam Lackorzynski, Jérémie Decouchant, Vincent Rahli, Francisco Rocha, and Paulo Esteves-Verissimo. Avoiding Leakage and Synchronization Attacks Through Enclave-Side Preemption Control. In *System Software for Trusted Execution (SysTEX)*, 2016.

[395] Shengye Wan, Jianhua Sun, Kun Sun, Ning Zhang, and Qi Li. SATIN: A Secure and Trustworthy Asynchronous Introspection on Multi-Core ARM Processors. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[396] Boyang Wang, Yantian Hou, Ming Li, Haitao Wang, and Hui Li. Maple: Scalable Multi-dimensional Range Search over Encrypted Cloud Data with Tree-based Index. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2014.

[397] Yongzhi Wang, Lingtong Liu, Cuicui Su, Jiawen Ma, Lei Wang, Yibo Yang, Yulong Shen, Guangxia Li, Tao Zhang, and Xuewen Dong. CryptSQLite: Protecting Data Confidentiality of SQLite with Intel SGX. In *International Conference on Networking and Network Applications (NaNA)*, 2017.

[398] Zhenghong Wang and Ruby B. Lee. Covert and Side Channels Due to Processor Architecture. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[399] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[400] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008.

[401] Xueqiang Wangy, Kun Sun, Yuewu Wangand, and Jiwu Jing. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[402] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2012.

[403] Andrew Waterman and Krste Asanovic. The RISC-V Instruction Set Manual Volume I: User-Level ISA. https://riscv.org/risc-v-isa/, 2017.

[404] Samuel Wedaj, Kolin Paul, and Vinay J. Ribeiro. DADS: Decentralized Attestation for Device Swarms. *ACM Transactions on Privacy and Security*, 22(3), 2019.

[405] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2018.

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

[407] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A Cache Timing Attack on AES in Virtualization Environments. In *Financial Cryptography and Data Security (FC)*, 2012.

[408] Peter Williams and Radu Sion. Single Round Access Privacy on Outsourced Storage. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2012.

[409] Johannes Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *ACM Workshop on Scalable Trusted Computing (STC)*, 2008.

[410] Songrui Wu, Qi Li, Guoliang Li, Dong Yuan, Xingliang Yuan, and Cong Wang. ServeDB: Secure, Verifiable, and Efficient Range Queries on Outsourced Database. In *International Conference on Data Engineering (ICDE)*, 2019.

[411] Lei Xu, Xingliang Yuan, Cong Wang, Qian Wang, and Chungen Xu. Hardening Database Padding for Searchable Encryption. In *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2019.

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

[413] Qiang Yan, Jin Han, Y Li, and RH Deng. A Software-Based Root-of-Trust Primitive on Multicore Platforms. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, 2011.

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

[415] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. Cryptology ePrint Archive. Report 2016/224, 2016. https://eprint.iacr.org/2016/224.pdf.

[416] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. ATRIUM: Runtime Attestation Resilient Under Memory Attacks. In *International Conference On Computer Aided Design (ICCAD)*, 2017.

[417] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y. Thomas Hou. CacheKit: Evading Memory Introspection Using Cache Incoherence. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[418] Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[419] 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)*, 2012.

[420] Zhangkai Zhang, Xuhua Ding, Gene Tsudik, Jinhua Cui, and Zhoujun Li. Presence Attestation: The Missing Link in Dynamic Trust Bootstrapping. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[421] Kai Zhao and Lina Ge. A Survey on the Internet of Things Security. In *International Conference on Computational Intelligence and Security (CIS)*, 2013.

[422] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

List of Acronyms

AES	Advanced Encryption Standard
AES-NI	AES New Instructions
AEX	Asynchronous Enclave Exit
AM	Acoustic Model
ASCI	Anti Side-Channel Interference
ASLR	Address Space Layout Randomization
ASR	Automated Speech Recognition
BIOS	Basic Input Output System
BTB	Branch Target Buffer
BYOD	Bring Your Own Device
CA	Certification Authority
CBC	Cipher Block Chaining
CFI	Control-Flow Integrity
CFG	Control-Flow Graph
CoT	Chain of Trust
COTS	Commercial Off-The-Shelf
CPI	Combined Probability of Inclusion
CPU	Central Processing Unit
CRL	Certificate Revocation List
CRT	Chinese Remainder Theorem
CSME	Converged Security and Management Engine
DBMS	Database Management System
DEP	Data Execution Prevention
DFI	Data-Flow Integrity
DFG	Data-Flow Graph
DMA	Direct Memory Access
DNA	Deoxyribonucleic Acid
DNN	Deep Neural Network
DOP	Data-oriented Programming
DoS	Denial-of-Service
DRBG	Deterministic Random Bit Generator
DRAM	Dynamic Random Access Memory

DRTM	Dynamic Root of Trust for Measurement
EA-MAC	Execution-Aware Memory Access Control
EA-MPU	Execution-Aware Memory Protection Unit
ECC	Elliptic Curve Cryptography
ECU	Electronic Control Unit
FHE	Fully Homomorphic Encryption
ELF	Executable Linking Format
EPC	Enclave Page Cache
FPGA	Field Programmable Gate Array
GCM	Galois/Counter Mode
GPS	Global Positioning System
GPU	Graphics Processing Unit
HE	Homomorphic Encryption
HMAC	Keyed-Hash Message Authentication Code
HMM	Hidden Markov Model
HPC	Hardware Performance Counter
IDT	Interrupt Descriptor Table
IEEE	Institute of Electrical and Electronics Engineers
IMV	Integrity Measurement Value
IPP	Integrated Performance Primitives
IoT	Internet of Things
IP	Intellectual Property
IP	Internet Protocol
IPC	Inter-Process Communication
IR	Intermediate Representation
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LA	Legacy App
LLC	Last Level Cache
LM	Language Model
LoC	Lines of Code
LOS	Legacy OS
LUT	Look-Up Table

MAC	Mandatory Access Control
MAC	Message Authentication Code
MCU	Microcontroller Unit
MDM	Mobile Device Management
MitM	Man-in-the-Middle
MMIO	Memory-Mapped Input/Output
MMU	Memory Management Unit
MPU	Memory Protection Unit
MSH	Multiset Hash
MSR	Model-Specific Register
NoC	Network-on-Chip
ORAM	Oblivious RAM
OS	Operating System
OTP	One-Time-Programmable
PC	Program Counter
PCI	Peripheral Component Interconnect
PCM	Performance Counter Monitor
PCR	Platform Configuration Register
PKI	Public Key Infrastructure
PMC	Performance-Monitoring Counter
PoC	Proof of Concept
PRNG	Pseudorandom Number Generator
PUF	Physical Unclonable Function
PvE	Provisioning Enclave
QE	Quoting Enclave
RA	Remote Attestation
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RNG	Random Number Generator
ROM	Read-Only Memory
ROP	Return-Oriented Programming
RoT	Root of Trust
RSA	Rivest–Shamir–Adleman
RTM	Root of Trust for Measurement

RTOS	Real-time Operating System
SA	Sanctuary App
SDK	Software Developer Kit
SGX	Software Guard Extensions
SHA1	Secure Hash Algorithm 1
SL	Sanctuary Library
SMC	Secure Monitor Call
SMC	Secure Multiparty Computation
SMM	System Management Mode
SMT	Simultaneous Multithreading
SoC	System-on-Chip
SRAM	Static Random Access Memory
STR	Short Tandem Repeats
SQL	Structured Query Language
SVM	Secure Virtual Machine
TA	Trusted App
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TF	Trusted Firmware
TLB	Translation Look-aside Buffer
TLS	Transport Layer Security
TOCTOU	Time-Of-Check-Time-Of-Use
TOS	Trusted OS
TPM	Trusted Platform Module
TSX	Transactional Synchronization Extensions
TXT	Trusted Execution Technology
TZASC	TrustZone Address Space Controller
UEFI	Unified Extensible Firmware Interface
UI	User Interface
VM	Virtual Machine
VMM	Virtual Machine Monitor
WSN	Wireless Sensor Network

Erklärung gemäß §9 der Promotionsordnung

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

Darmstadt, Germany, April 2020

Franz Ferdinand Peter Brasser