Master's Programme in Security and Cloud Computing

Towards attack-tolerant trusted execution environments

Secure remote attestation in the presence of side channels

Max Crone

Aalto University

MASTER'S THESIS

Aalto University — KTH Royal Institute of Technology

MASTER'S THESIS 2021

Towards attack-tolerant trusted execution environments

Secure remote attestation in the presence of side channels

Max Crone

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, 12 July 2021

Supervisors: Prof. N. Asokan
Prof. Panagiotis Papadimitratos
Advisors: Dr. Hans Liljestrand
Dr. Lachlan Gunn

Aalto University
School of Science

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science

Master's Programme in Security and Cloud Computing

Aalto University

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Abstract

Author
Max Crone

Title
Towards attack-tolerant trusted execution environments: Secure remote attestation in the presence of side channels

School School of Science

Master's programme Security and Cloud Computing

Major Security and Cloud Computing **Code** SCI3113

Supervisors Prof. N. Asokan, Prof. Panagiotis Papadimitratos

Advisors Dr. Hans Liljestrand, Dr. Lachlan Gunn

Level Master's thesis **Date** 12 July 2021 **Pages** 64 **Language** English

Abstract

In recent years, trusted execution environments (TEEs) have seen increasing deployment in computing devices to protect security-critical software from run-time attacks and provide isolation from an untrustworthy operating system (OS). A trusted party verifies the software that runs in a TEE using remote attestation procedures. However, the publication of transient execution attacks such as Spectre and Meltdown revealed fundamental weaknesses in many TEE architectures, including Intel Software Guard Exentsions (SGX) and Arm TrustZone. These attacks can extract cryptographic secrets, thereby compromising the integrity of the remote attestation procedure.

In this work, we design and develop a TEE architecture that provides remote attestation integrity protection even when confidentiality of the TEE is compromised. We use the formally verified seL4 microkernel to build the TEE, which ensures strong isolation and integrity. We offload cryptographic operations to a secure co-processor that does not share any vulnerable microarchitectural hardware units with the main processor, to protect against transient execution attacks. Our design guarantees integrity of the remote attestation procedure. It can be extended to leverage co-processors from Google and Apple, for wide-scale deployment on mobile devices.

Keywords trusted execution environment, remote attestation, sel4, microkernel, arm trustzone, intel sgx, side-channels, transient execution attacks

Preface

This thesis work has been conducted as a conclusion to the Erasmus Mundus Master's degree programme in Security and Cloud Computing (SECCLO). I am most grateful for the time I have been able to spend with my SECCLO friends while learning, laughing, eating, and talking. Those moments have taught me more than I had ever envisioned, and I will carry all of you forth in my mind and in my stories.

I want to thank my supervisor **Prof. N. Asokan** from Aalto University for his guidance throughout my research. I am additionally very grateful for the amazing discussions I have had with my advisor **Dr. Hans Liljestrand** from University of Waterloo. There is much I have come to learn and appreciate on a more fundamental level because of the motivation and insight from Hans. Furthermore, I would like to thank my supervisor **Prof. Panagiotis Papadimitratos** from KTH and my second advisor **Dr. Lachlan Gunn** from Aalto University.

Many more people made my time with the Secure Systems Group at Aalto University one of fond memories. For the sake of brevity, my warmest thanks to **Niina Idänheimo**, our collective and loving "mom".

During the work that lead to this thesis, I was supported by scholarships from the Helsinki-Aalto Institute for Cybersecurity (HAIC). This financial support is gratefully acknowledged.

A special thanks to my friends **Felix Maurer**, for our endless discussions and explorations, and **Jack Henschel**, for all our outdoor gym sessions, sauna evenings, and the many adventures.

I thank my parents, **Ton** and **Ellen**, for their love, encouragement, and support. And my brother, **Thijs**, for the sense of life we developed together.

Espoo, July 6, 2021,

Max Crone

Contents

List of Figures

List of Abbreviations

ADL architecture description language.

API application programming interface.

BTB branch target buffer.

capDL capability description language.

CNode capability node.

CSpace capability space.

DRM digital rights management.

EL exception level.

ELF executable and linkable format.

EPC enclave page cache.

HSM hardware security module.

IPC inter-process communication.

ISA instruction set architecture.

LLC last-level cache.

LVI load value injection.

MAC message authentication code.

MEE memory encryption engine.

NS non-secure.

OEM original equipment manufacturer.

OS operating system.

PD page directory.

PHT pattern history table.

PKI public key infrastructure.

PoC proof-of-concept.

PSP platform security processor.

PT page table.

REE rich execution environment.

ROB re-order buffer.

RoT root of trust.

RPC remote procedure call.

RSB return stack buffer.

SCR secure configuration register.

SECS SGX enclave control structure.

SGX Software Guard Exentsions.

SMC secure monitor call.

SoC system on chip.

STL store to load.

TA trusted application.

TCB trusted computing base.

TEE trusted execution environment.

TLV tag-length-value.

TPM trusted platform module.

VSpace virtual address space.

1. Introduction

"Nobody ever figures out what life is all about, and it doesn't matter. Explore the world. Nearly everything is really interesting if you go into it deeply enough"

Richard P. Feynman

The development of the Internet and its increasing presence in society have spurred demand for computer systems. Computers are now found everywhere, from large scale industrial data centers, to tiny applications in embedded environments, and mobile devices everyone carries. With the increased reliance on computing devices has also come an increase in requirements for their security.

Already since the 1960's, there have been efforts to secure computer operating system (OS) against criminal actors [44]. Early work on security and standards mostly came from governmental organizations in the United States [22], including the Department of Defense (DoD) and the Air Force Electronic Systems Division. The resource-sharing computer systems of that time relied on separation enforced by the OS. Generally, these systems were found to provide practically no protection from deliberate attempts to violate the OS [22]. As a reaction, James Anderson and others introduced the concept of a reference monitor in a technical report from the 1970's [4], kicking off the serious attention to systems security in the IT sector. The developments of these times are captured in the proliferation of general security principles still applied to OSs today.

OS security in this thesis encompasses three principles: integrity, isolation, and access control.

The first principle is the *integrity* of computer programs and data stored in memory and other media, i.e., the bits that encode them stay consistent over time with high reliability. Violations of integrity, either malicious or unintentional, can cause significant problems. For example, the disfunction of software crucial for proper operation of a modern car might lead to a disaster. As another example, violations of data integrity in global financial systems can severely impact the life of real people.

The nature of the early multi-user systems and the concurrent operation of modern computers give rise to the second security principle; *isolation*. Even though an OS might be executing many different processes concurrently, none should be able to interfere with each other in an unauthorized manner. To securely run software, processes

must be isolated and prevented from affecting each other's memory and execution, except through well-defined and intended interfaces.

Then there is the security principle of *access control*. The OS is tasked with separating the data and other resources owned by different programs or users. It must only allow access to data for the entity that owns it, or for others who have been assigned appropriate rights. Access control is thus a method of providing confidentiality of resources.

Though the security principles are clear, in practice it has proven to be challenging to ensure complete adherence to them. OSs and other computer programs are so complex that formal verification of security properties never seemed feasible. Over time, security issues have become more relevant and of higher impact.

1.1 Breaking security

Memory safety bugs are prevalent and form one of the most dangerous types of software vulnerabilities [80]. Any such vulnerability is a potential information leak. They are found — among other places — in web browsers such as Chrome and Firefox, and cryptographic libraries protecting internet connections [80, 25]. The consequences of these bugs extend to the compromise of secret keys that protect passwords, banking details, and personal information sent over the internet.

Other run-time attacks exploit software defects to influence a targeted computer system into behaving maliciously instead of only leaking information. These might ultimately give the attacker access to the computer system. A famous example is the Morris worm [66], which exploited a software defect to spread itself to a large share of the computers connected to the Internet.

A different class of attacks are *side-channel* attacks. Side-channel attacks exploit information retrieved from computer system components functioning as intended, such as timing information [14, 32], power consumption [49, 82], and electromagnetic leaks [2]. These attacks often aim to retrieve cryptographic keys or other secrets, that would allow access to confidential data [14] or enable impersonation [34].

1.2 Trusted execution

Due to the prevalance of security vulnerabilities, researchers and programmers have sought solutions to protect security-critical software against attacks. One idea is to separate sensitive software and data from the OS. Compromised applications or even an adversary controlling the OS can then no longer access the sensitive software. This is

the premise of a trusted execution environment (TEE). A TEE isolates the execution of confidential programs and operations on sensitive data from a potentially malicious OS. It does this through a combination of software and hardware mechanisms. Examples of technologies for TEEs are Arm TrustZone [6] and Intel SGX [57].

However, the existing TEE solutions have not been infallible either. Researchers identified more than a hundred vulnerabilities in TrustZone-assisted TEE systems in recent years [17]. There are more than twenty published attacks on Intel SGX [61], some of which manage to compromise the full memory contents of the SGX system. Many of these are side-channel attacks that compromise the secure keys used for attestation in these systems. This consequently breaks the trust in the software running in these TEEs.

1.3 Contributions

We present an architecture for TEE remote attestation that upholds integrity guarantees in the presence of the same attacks that compromised earlier TEE technologies. We achieve this by leveraging the formally verified seL4 microkernel and a secure co-processor for cryptographic operations. The following lists our contributions.

- Designed and implemented a TEE remote attestation architecture with strong integrity and isolation security guarantees in the presence of vulnerabilities and side-channel attacks.
- Designed a method to dynamically load and execute user-provided trusted applications (TAs) in our TEE.
- Considered the performance overhead of our TEE remote attestation architecture.

1.4 Structure of the thesis

The remainder of this thesis is structured as follows. Chapter 2 presents a background on TEEs, side-channel attacks, and the seL4 microkernel. Chapter 3 formulates the system and adversary model and makes the security goals and performance goals of this work explicit. Chapter 4 describes our proposed TEE remote attestation system design. Chapter 5 explains how we implemented our TEE remote attestation system. In Chapter 6, we evaluate our system's implementation from a security perspective and consider its performance properties. Chapter 7 discusses what we would have done differently given more resources and describes future work. In Chapter 8, we describe the related work with respect to TEE designs and defenses. Finally, Chapter 9 concludes this thesis work.

2. Background

"Let's start with the end of the world, why don't we? Get it over with and move on to more interesting things."

N.K. Jemisin, *The Fifth Season*

2.1 Trusted execution environments

A trusted execution environment (TEE) is an environment that enforces that code and data loaded inside cannot be tampered with or read by code outside that environment. TEEs provide isolation for small pieces of trusted software from the rest of the device, called the rich execution environment (REE), which typically includes an operating system (OS), such as Linux, and user space applications. To enforce this isolation, TEEs use hardware-based security mechanisms in combination with software security mechanisms. This allows developers to design applications that remain secure even when the REE OS is compromised.

Trusted platform modules (TPMs) and hardware security modules (HSMs) are two examples of TEEs. A TPM generally provides a secure key storage for hard drive encryption and a minimal set of cryptographic operations often used for validating steps in the booting processing. An HSM typically provides hardware-accelerated cryptographic operations and secure key and certificate storage. Both are often implemented as separate hardware microcontrollers, embedded at different levels of a system. In this work, we are concerned with a TEE that provides an isolated execution environment on the main processor, where sensitive data is stored, processed, and protected. Specifically, by running it on the main processor it takes advantage of higher processing speed and larger amounts of memory than other secure environments — i.e., TPMs and HSMs.

There are multiple use cases for TEEs [26, 68].

Digital rights management (DRM) solutions might make use of a TEE, when distributors of media content (e.g., films, music) want to protect their digitally encoded content from being copied by the owner of the device. For example, a TEE decrypts an incoming media stream and pushes it over a trusted path to an approved display. Or, a TEE prevents other applications on the device from reading the copyright-protected contents when they are stored on the device. It does this by only exposing the content

to environments approved by the distributor or developer. As an example, Widevine is a content protection system by Google that specifies the use of a TEE [93].

A TEE is also suitable for digital payment scenarios that require high security and trust in the device. It helps to provide strong identification and proof of transaction. The TEE secures the execution of cryptographic algorithms used for signature verification and authentication.

Many devices today secure their biometric authentication methods with a TEE, e.g., Apple's and Samsung's mobile devices [5, 85]. The TEE stores a template for the biometric factor — face, or fingerprint — and performs extraction and matching of new samples in a secure manner. They often incorporate trusted hardware paths for input, to minimize the opportunities for attackers to gather data or exploit the user via malicious user-interface elements.

In the scenario of cloud computing, a tenant can use TEEs to store sensitive data such as customer details or to perform sensitive computations, while preventing the cloud provider from accessing the data. This provides better data security for the tenant and reduced liability for the cloud provider.

2.1.1 Security requirements and capabilities

The GlobalPlatform industry consortium defines a set of high level security requirements for TEEs in their standard on TEE system architecture [84]. We summarize those requirements here.

- Protect assets from the REE and other (execution) environments, through hardware mechanisms that other environments cannot control.
- Prevent system components from accessing assets in a TEE unless they themselves are a protected asset of the TEE.
- Protect against some physical attacks [83].
- Instantiate the trusted OS from a root of trust (RoT) through a secure boot process.
- Provide trusted storage of data and keys.
- Enforce that software outside the TEE can only call the external application programming interface (API) of the TEE, that can verify the acceptability of an operation. Prevent outside software from directly calling functionality from any of the TEE's internal APIs.

To prevent hardware simulation, a TEE uses a hardware RoT — a highly reliable, secure-by-design component that performs specific, security-critical functions. The root of trust consists of a set of private keys that are provisioned to the device during

manufacturing, in a manner that prevents them from being changed. Manufacturers store the respective public keys, together with information from trusted parties (i.e., chip vendors). Chip vendors use this key infrastructure to, for example, sign and distribute trusted firmware.

Generally, we consider an environment to be a TEE when it has three main capabilities.

1. **Isolation** of execution from the REE, providing confidentiality and integrity for code and data.
2. **Secure storage** of persistent data, guaranteeing confidentiality and integrity across reboots in the presence of an adversarial REE.
3. **Attestation** of the software running in the TEE; attesting to a third party that a certain version of the application is loaded or executing in the TEE.

Remote attestation

Remote attestation is a process of verifying the integrity of a remote system's state. This generally happens via a challenge-response protocol, where the relying third party poses a fresh challenge that the remote system responds to with a verifiable cryptographic signature.

The remote system additionally includes an attestation of the software on the system. An attestation is a cryptographic signature that certifies a hash of the software's state. Though there exist control-flow attestation mechanisms that take place later in the execution lifecycle of a software application [1], in this work, we take attestation to mean the verification of the initial state of an application. This is called static attestation.

Remote attestation requires a trust assumption for authenticity of the attestation report. The platform of the remote system thus generally contains a RoT that forms the basis of a remote relying party's trust in a system and, consequently, their trust in the remote attestation procedure. The relying party can verify an attestation report against an endorsement certificate created by the trusted hardware's manufacturer.

2.1.2 General architecture

The GlobalPlatform industry consortium specifies a software architecture for TEEs that outlines the relationship between major components. Figure 2.1 depicts a simplified architecture based on GlobalPlatform standards [84].

The trusted applications (TAs) in a TEE generally do not run complete user-facing applications. Instead, normal applications in the REE invoke them via client applications to perform specific tasks that require the security guarantees provided by the TEE. For

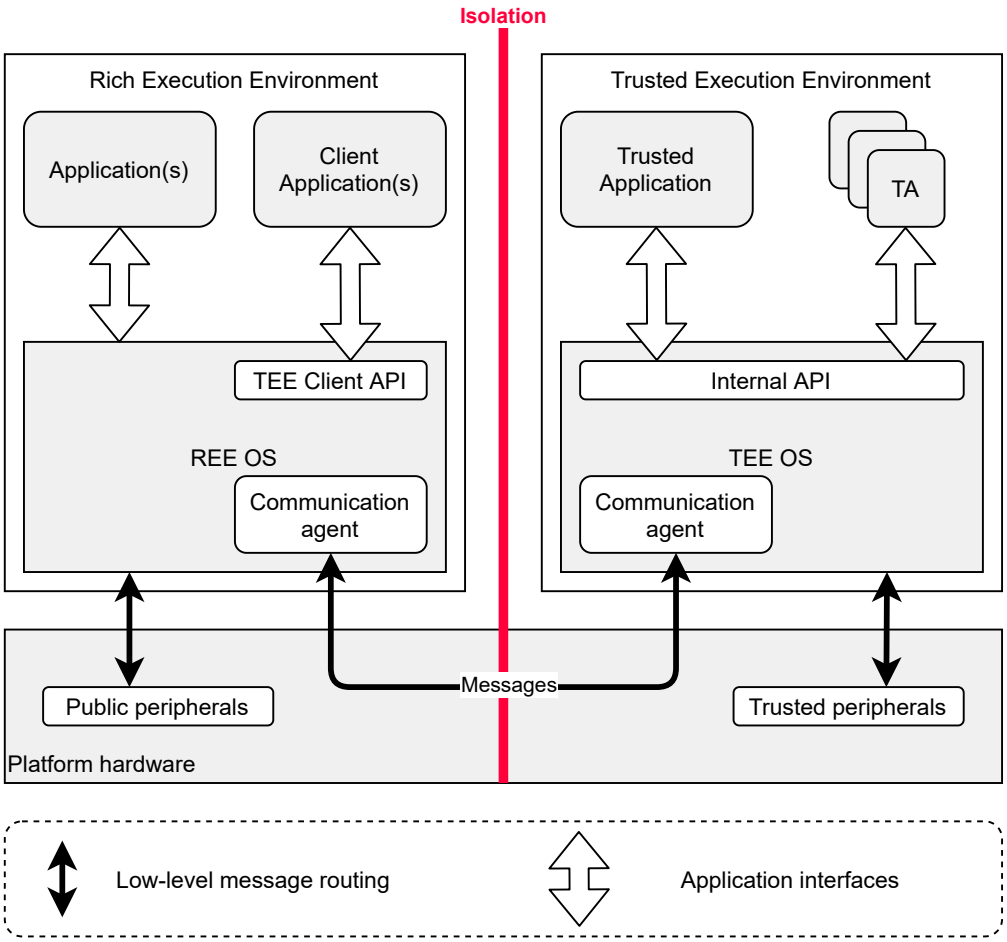

Figure 2.1. General trusted execution environment (TEE) software architecture. The TEE and rich execution environment (REE) are isolated by hardware mechanisms, but can still communicate via the platform application programming interface (API). The TEE runs its separate, trusted operating system (OS), and exposes internal APIs to provide functionality to every trusted application (TA) it runs.

example, a banking application might only call a TA to handle user authentication when an individual starts a transaction. The REE application can manage the rest of the transaction process that does not require those higher levels of security.

2.1.3 Real-world architectures

There exist several hardware technologies that support the implementation of TEEs. Arm designed TrustZone as extension to a selection of their architectures, which they license to other companies. Microprocessor manufacturers AMD and Intel developed platform security processor (PSP) and Software Guard Exentsions (SGX) respectively. For RISC-V, there are multiple TEE designs available [58, 54, 29]

Intel SGX and Arm TrustZone are the most widely used in TEE implementations today.

Arm TrustZone

Arm TrustZone is a set of security extensions to the Arm architecture [69]. TrustZone adds processor support for a new execution mode and memory protections that allow

the system to distinguish between two protection domains called secure world and non-secure world. Both worlds are hardware-isolated and have different sets of privileges, with non-secure software prevented from accessing any resources from the secure world. At any point in time, the processor operates exclusively in one of the two worlds.

The current world in which the processor executes is encoded in a new processor bit called the non-secure (NS) bit, which can be read from the secure configuration register (SCR). TrustZone introduces the monitor mode which is responsible for preserving the processor state when world switches occur. Via a new privileged instruction called the secure monitor call (SMC), software stacks in both worlds can be bridged by the secure monitor software. TrustZone also provides memory extensions to configure memory regions as secure or non-secure, to protect non-secure world applications from accessing memory belonging to the secure world.

The Arm architecture has different levels of access to system and processor resources called exception levels (ELs) [8]. The common usage model of ELs is as follows.

- EL0 for applications
- EL1 for the OS kernel
- EL2 for hypervisors
- EL3 for the secure monitor

The secure monitor that handles the world switches operates on EL3, which is the highest privilege level. It has thus access to more system and processor resources than any OS, which allows it to effectively isolate the OS in the non-secure world from the OS in the secure world.

This described the TrustZone extensions for Arm's application processors (Cortex-A). The TrustZone requirements for Arm microcontrollers (Cortex-M) [7] are different, because of the processor family's focus on fast context switches and low-power applications. TrustZone for Armv8-M uses a memory-map based division between worlds where transitions take place in exception handling code. This allows it to remove the monitor mode and thus the need for any secure monitor software, leading to more efficient world transitions.

TrustZone is not a TEE in itself yet. Instead, its secure world provides the execution environment where a TEE may reside. Examples of TEEs implemented on top of TrustZone technology are Samsung Knox [70] and Open-TEE [56].

During boot, a TrustZone-enabled Arm processor enters the secure world to allow privileged software to initialize their protections and perform hardware measurement operations. This feature enables the implementation of a secure boot process that veri-

fies the platform state, which forms the basis for trusted remote attestation. For example, Samsung Knox provides secure boot using TrustZone in this manner [69].

Intel SGX

Intel SGX differs from TrustZone in the size of its trusted computing base (TCB). As a general rationale, a smaller TCB has a smaller attack surface and thus a lower chance of vulnerabilities, leading to better security. The split-world architecture of TrustZone means that any software operating at the higher privilege level is inherently part of the TCB, because they have greater access to the machine. In contrast, SGX uses an *enclave* architecture, backed by trusted hardware, that provides strong isolation between TAs, such that a trusted OS does not become part of the TCB.

Enclaves are execution environments for code and data that are protected from any outside software environment. The threat model for enclaves only trusts the enclave itself and treats all outside processes — including the OS and any hypervisor — as potential adversaries.

An SGX enclave is part of the host process application. However, an application is not able to access an enclave's memory, which resides in the enclave page cache (EPC). The EPC is a protected area of memory consisting of 4 KB pages that stores enclave code and data. This memory area is encrypted using a new hardware unit called the memory encryption engine (MEE), and is only decrypted once inside the physical processing core [37].

An application creates and initializes enclaves for its own use. SGX associates every enclave with an SGX enclave control structure (SECS) that stores metadata in a dedicated EPC page. The initialization of an SECS creates a new enclave. An application then loads the initial code and data into the enclave. For every page loaded, the system software updates the enclave's measurement that is used for attestation. Measurements are cryptographic hashes that uniquely identify a system state. An application initializes its enclave via Intel's architectural *Launch Enclave*. It can then enter the enclave by transferring control to a pre-determined location within it.

SGX supports two types of attestation: local and remote [3, 45]. In local attestation, an enclave proves its identity to another target enclave on the same system. It produces a signed attestation report that contains a hash of the measured contents of the enclave (`MRENCLAVE`), its sealing identity (`MRSIGNER`), its attributes, and additional information. It also appends a message authentication code (MAC) created with a symmetric key unique to the platform. The target enclave verifies this report to establish confidence that it is communicating with a legitimate enclave on the system.

Where local attestation uses a symmetric key for verification, remote attestation to

outside the platform requires asymmetric cryptography. SGX provides an architectural enclave specifically for remote attestation; the *Quoting Enclave*. The Quoting Enclave verifies reports from other enclaves on the system following the method described above. It then replaces the MAC with an attestation signature produced with the Attestation Key. The Attestation Key is generated by the Provisioning Enclave in SGX and Intel's provisioning service when a new enclave is created. The Quoting Enclave uses this key to generate attestations — called *quotes* — that can be verified to originate from a legitimate SGX enclave hosted on trusted hardware. It additionally includes the hashed measurements of the contents of the enclave. Finally, the remote party verifies that this value agrees with what they expected; ensuring that a specific piece of software is running.

The provisioning enclave is part of the trust chain on which attestation builds. During manufacturing, Intel generates a provisioning secret for a processor. Every processor additionally has secrets sealed into its e-fuses — a form of one-time programmable read-only memory. The provisioning enclave uses both elements to demonstrate the SGX platform's authenticity to Intel, in order to receive appropriate attestation keys. This ensures the trust in the later remote attestation phases.

2.2 Side-channel attacks

Side-channel attacks are a type of security exploit. Many classes of security exploits target specific vulnerabilites in software. Vulnerabilities such as memory safety violations, input validation errors, and race conditions are often caused by mistakes or inaccuracies from the people developing the software. In contrast, side-channel attacks are based on information gained from legitimate operation of computer systems. However, no one had expected this information could also be used to infer valuable data; it is thus *accidental* leakage of sensitive data. Originally, side-channels were understood to be the exploitation of physical phenomena that were the by-product of the execution of tasks on electronic devices. These original channels include power consumption, electromagnetic radiation, heat, and noise [79]. Researchers have shown how to find secret keys by analyzing power consumption [49], how to break power analysis countermeasures and extract compromising information via electromagnetic radiation [2], and how to extract RSA keys using acoustic cryptanalysis [34]. Timing attacks are another side-channel attack. They enable an attacker to infer secret information based on the time it takes a computer to perform certain tasks [14]. Timing attacks often target cryptographic operations to extract secret keys.

TEEs are also vulnerable to these side-channels. In 2017, a TEE based on TrustZone

was found to be vulnerable to a timing attack that enabled attackers to extract cryptographic secrets [17]. The CLKscrew attack exploits energy management mechanisms to extract secret cryptographic keys from TrustZone and escalate its privileges by loading self-signed code [82]. Plundervolt is a similar attack, but it targets Intel SGX instead of TrustZone [59].

These side-channel attacks are distinct from earlier computer security problems. As such, they also require different mitigations in comparison with the usual software vulnerabilities. These mitigations range from signal size reduction, time randomization, to the introduction of noise to decrease the amount of information that is leaked, and constant-time operations [79].

In the remainder of this section we describe the class of side-channels that appears most frequently in attacks against TEEs: microarchitectural side-channels. In particular, we consider cache side-channel attacks and transient execution attacks. These side-channel attacks exploit complex microarchitectural behavior used to improve processor performance.

2.2.1 Background on microarchitecture

A microarchitecture is the implementation of an instruction set architecture (ISA) in a processor. The ISA serves as an interface between hardware and software. As such, it abstracts over details concerning functional implementation, such as pipelines and caches. These constitute the microarchitecture instead. Examples of ISAs are the ubiquitous x86 architecture [9], Armv8 [8], and more recently RISC-V [92]. Examples of microarchitectures on the other hand, are Intel's Cypress Cove implemented on the eleventh generation Intel Core desktop microprocessors [42], and Arm's Cortex-A78 implemented in several Samsung Exynos [28] and Qualcomm Snapdragon microprocessors [30], among others. Both the architecture and microarchitecture are stateful. The architectural state includes memory and registers that are accessible to the programmer. Whereas the microarchitectural state includes entries in various caches that are used to improve performance without affecting the architectural state.

Instruction cycles and pipelining

The instruction cycle is an important concept of microarchitectures. By repeating the instruction cycle, the processor runs programs. On a high level, the instruction cycle consists of four steps.

1. Fetch and decode an instruction
2. Retrieve data needed for the instruction
3. Execute the instruction
4. Write the results to the destination register

A great improvement to performance of the cycle is the concept of instruction pipelining. Early processors would execute all four steps sequentially for a single instruction before they moved on to the next instruction. This approach is inefficient. While executing an instruction, the decoding circuitry of the processor is idle. In fact, with this sequential approach, large parts of the processor circuitry will be idle most of the time. It would instead be much more efficient to already decode the next instruction while processing the current one. This is exactly the idea of instruction pipelining. In order to keep every part of the processor busy, incoming instructions flow through the processor in multiple stages. Thus at any clock cycle, there can be multiple instructions in the pipeline. The aim is to keep every part of the processor busy. Modern pipelines consist of fourteen to twenty stages. Embedded processors might do with less; between six and twelve. Putting significantly more stages in the pipeline eventually hits the law of diminishing return. This is exemplified by Intel's Prescott microarchitecture that had a deeper instruction pipeline of 31 stages, but experienced issues with power consumption and heat dissipation [43].

There are several more techniques for optimizing instruction pipelines. However, these optimizations also introduce new vulnerabilities such as side channels, e.g., when instructions alter the cache state in a way that leaks sensitive data. Such vulnerabilities are hard to mitigate, because programmers cannot directly control the microarchitectural state, and the microarchitectural behavior is often complex, propriety, and undocumented.

Out-of-order execution

Processors optimize their utilization by allowing for *out-of-order* execution of instructions in the pipeline. For example, performing a memory operation takes a relatively long time and if the processor were only executing instructions in-order, it would have to wait until the memory operation returned. Instead, the processor might already execute the next instruction *out-of-order* while it is waiting. This speeds up general execution of programs.

A processor stores instructions it completed out-of-order in a buffer. When the execution flow catches up, the processor retrieves the instructions from the buffer in the correct order. Only then do the results of these instructions become visible in the architectural state.

Speculative execution

When performing out-of-order executions, a processor often does not know the future instruction sequence yet. For example, this is the case when execution reaches conditional branch instructions whose direction depends on earlier instructions that have not completed yet. Recall the example of a memory operation from earlier. To address

this problem, processors speculate on the value that determines a future instruction sequence. They then speculatively execute instructions along that path. When execution of the speculated instruction is resolved, the processor verifies whether the prediction was correct. Upon a correct prediction, the processor retrieves the speculative instructions' results and applies them in program execution order. Otherwise, the processor abandons pending instructions along the path, restores its state from a checkpoint, and resumes execution along the correct path. The speculative execution technique increases the performance of microprocessors.

Branch prediction

Branch prediction is related to the instruction pipelining concept. Imagine that one of the instructions going through the pipeline is a branch instruction. A branch instruction is the implementation of an if-else construct. Depending on the argument value, there will be different instruction sequences that come next. However, when the branch instruction is finally executed, several other instructions will already be in the pipeline. If the instructions in the pipeline are from the incorrect branch, the one that does not end up being followed, the processor wasted cycles on executing them anyway. This is where the *branch predictor* comes into play. Its task is to make the most accurate predictions for which branch is most likely to be taken. This prediction will then inform which instruction sequence flows through the pipeline. The branch predictor bases its guesses on historical information, e.g., which branch has been taken more often recently. It is built as a digital circuit on the processor.

Branch mispredictions have a larger effect on system performance when the instruction pipelines are deeper. Because instructions cannot just be removed from the pipeline, they have to go through all stages up until execution. The number of cycles wasted upon a branch misprediction is thus equal to the number of stages from fetch to execution.

Because the branch predictor acts on historical information, it can also be intentionally *trained* to behave in a specific way. A malicious actor could use this to either infer what computations have altered its behavior, or even to transiently execute an attacker-chosen instruction sequence.

Caches

Processor caches are an important part of the memory hierarchy in modern computers [81, 32]. With the speeds of memory and processors diverging over the last decades, caches help to bridge the gap. Caches are fast, but small and expensive, memory that are used to store data being accessed by the processor. Most processors include a hierarchy of cache levels. Today, they often have three levels: from L1, which is fastest and smallest, to L3, which is larger but slower. The L1 cache is frequently split up in

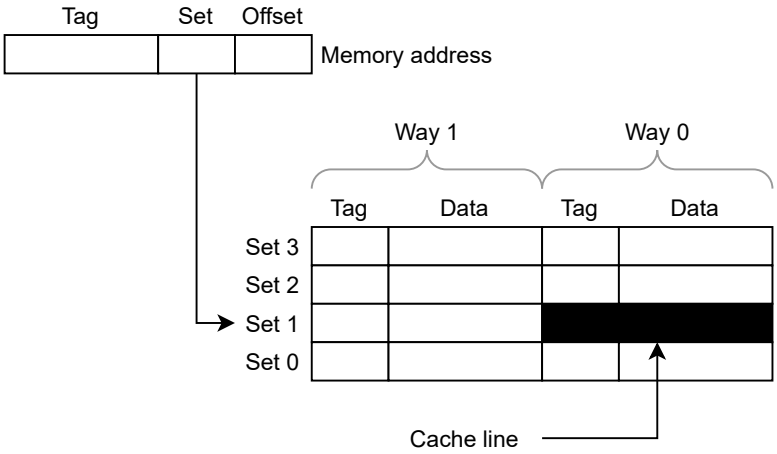

Figure 2.2. Typical addressing of a cache line in a 2-way set-associative cache. The memory address determines the exact mapping. It is split up in different *cache fields* that indicate a byte offset, the set index, and the identifying tag (consisting of the remaining most significant bits).

separate caches for instructions and data, called L1i and L1d respectively. The use of caches is more efficient when the *hit rate* is high, i.e., when a higher fraction of requests is satisfied from the cache instead of accessing the slower main memory. A cache miss occurs when the requested data is not found in any of the caches. The processor then fetches the data from main memory and places it in a cache. If there already exists other data in the slot, the processor evicts — i.e., removes — it before storing the new data. A higher share of cache evictions impacts performance negatively.

Caches are divided into *cache lines* [39]. A cache line holds an aligned block of adjacent bytes from memory, usually a power of two in size. Grouping together adjacent bytes makes use of spatial locality and limits implementation complexity. As a consequence, if replacing any single byte in the cache, the whole cache line needs to be evicted.

A cache placement policy determines where the data at a memory address is placed in a cache. Most architectures specify a set-associative policy. A W-way set-associative cache divides its memory into *cache sets* that each consist of W cache lines. Cache sets are directly addressable, but the *way* where the cache line resides must be found by comparing its tag. Figure 2.2 illustrates a typical cache addressing scenario.

2.2.2 Cache-based side-channel attacks

Many cache-based side-channels originate from timing differences measurable by the attacker. The intuition stems from the fundamental purpose of caches; the processor wants to access certain data faster. An attacker can measure memory access times to infer whether the victim has touched the corresponding cache lines. When a measurement takes relatively long (~100ns), the attacker infers that the data had to come from

memory rather than from cache. In contrast, if the request returned quickly (10-25ns), the attacker knows that it hit the cache. This principle allows an attacker to learn about the memory access patterns of the victim. Researchers have proposed several techniques to exploit this.

PRIME+PROBE

This technique from 2006 [67] consists of two main steps. Firstly, an attacker *primes* an area of the cache by filling cache sets with its own cache lines. After waiting for the victim to have executed again, the second step encompasses *probing* — accessing and changing data — the cache lines the attacker loaded earlier. By observing the timing of each probe, the attacker sees which cache lines have been evicted. This tells them that the victim must have accessed an address in memory that maps to the same cache set.

FLUSH+RELOAD

The advantage of FLUSH+RELOAD over PRIME+PROBE is that it allows to target a specific cache line rather than only a cache set [95]. The technique uses the last-level cache (LLC), which is shared by all cores on a processor implementing the x86 architecture, and it relies on the existence of shared virtual memory (e.g., shared libraries).

A FLUSH+RELOAD attack consists of three phases. Firstly, the attacker flushes the targeted line from the cache. Secondly, they wait to allow for the victim to access the memory line. Thirdly, the attacker reloads the memory line and measures the time it takes. If the victim accessed the memory line and caused it to appear in the same cache location again, the reload operation will return quickly. Otherwise, the victim did not access the memory line and the attacker's probe will take longer because the line needs to be fetched from main memory first.

FLUSH+RELOAD is a generic technique also used as a building block for more advanced attacks, such as Spectre and Meltdown.

2.2.3 Transient execution attacks

Transient instructions are instructions executed speculatively by the processor, but that are discarded after a pipeline flush, e.g., on a branch misprediction [16]. The pipeline flush discards any architectural effects of these pending instructions. However, transient execution attacks leverage the fact that transient instructions have persisting microarchitectural side effects to exfiltrate data across security boundaries. An attacker can steer or even choose the instruction sequences to execute transiently, depending on the attack they employ.

The field of transient execution attacks emerged recently with the publication of the

Spectre and Meltdown attacks [50, 55]. These attacks have shown how to dump memory, intercept passwords, and reconstruct images. Transient execution attacks are also shown to work against TEEs, e.g., in SgxPectre [19] and Foreshadow [15].

Spectre

Spectre attacks trick a victim into speculatively executing instructions which leak confidential data via a covert channel [50]. These attacks consist of two main parts: a microarchitectural element to exploit, and a covert channel through which to exfiltrate data. Canella et al. categorize Spectre variants based on the microarchitectural elements they exploit: pattern history table (PHT), branch target buffer (BTB), return stack buffer (RSB), and store to load (STL) [16]. All of these are part of predictive optimizations in microprocessors. Spectre attacks use cache-based covert channels (Section 2.2.2) to exfiltrate data across architectural security boundaries. The original paper describes attacks using Flush+Reload and a variant called Evict+Reload [50].

Spectre attacks bypass *software-defined* security boundaries such as bounds checking. An attacker can trick the victim into transiently executing instructions on memory locations they have access to, but which the attacker is not authorized to address directly.

To trick the victim into speculative execution of these instructions, an attacker first prepares the processor by mistraining its prediction machinery. The attacker also prepares the covert channel, e.g., by performing the flush portion of the Flush+Reload attack. As a final step in the setup, the attacker performs operations to trigger the desired speculative state. For example, they perform targeted memory reads that cause the processor to evict a specific value from cache that is needed to determine a branch target.

In the second phase, the processor performs the transient execution of instructions. The transient execution may be triggered by requesting the victim to perform an action via a system call, socket, or file. The results of this speculative execution reflect in the prepared microarchitectural side-channel. For example, the victim speculatively reads a memory value at an attacker-chosen address and subsequently performs a memory operation that exposes the value via its modifications to the cache state.

Lastly, the attacker recovers the confidential data by timing the reads of memory addresses for the monitored cache lines, similar to a traditional side-channel attack.

Meltdown

Whereas Spectre attacks exploit transient execution following mispredictions of control or data flow, Meltdown exploits transient execution after a faulting instruction [55]. The key observation is that speculation may still occur after exceptions in transient

instructions. This characteristic allows the Meltdown attack to read kernel memory from user space, bypassing architectural isolation barriers.

Meltdown is similar to Spectre and as such consists of the same two abstract parts again. Firstly, Meltdown uses exceptions as the element to cause transient execution. For the second phase, Meltdown uses a cache-based covert channel.

Most Meltdown variants rely on page faults to trigger the transient execution [16]. The original attack reads data from a kernel memory address, which eventually causes a page fault. Because of out-of-order execution, vulnerable processors identify the exception generating instruction, but only tag its entry into the re-order buffer (ROB) with an exception bit. They then continue speculative execution of other instructions. The exception is handled once instructions are retired from the ROB and become visible in the architectural state. By that point, transient instructions encoded the privileged data into the microarchitectural state.

In its second phase, Meltdown uses the Flush+Reload technique to exfiltrate confidential data kernel memory. Transient instructions encode the privileged data by accessing specific cache lines. The attacker recovers this value by measuring the access time for all cache lines.

All major operating systems used to map the kernel address space into the virtual address space of every process. They also mapped physical memory into the kernel address space. As such, an attacker employing Meltdown could read the entire physical memory of a computer system. Since publication of the Meltdown attack, mitigations isolate the kernel page table [35].

Extensions targeting Intel SGX

The original Spectre and Meltdown attacks were not demonstrated on Intel SGX. However, later research has shown variants that target Intel SGX. Consequences include the keypair extraction of Intel's architectural enclaves (Section 2.1.3) and the compromise of enclave secrets.

Foreshadow is an extension of Meltdown that targets Intel SGX [15]. It leverages the same processor vulnerability that allows an attacker to use the results of transient unauthorized memory accesses. Meltdown exploits transient execution to access kernel memory before the fault caused by the access violation is handled and Meltdown triggers a race condition by causing a page fault after accessing kernel memory. This enables the exploitation of transient instructions, before the fault handler causes them to revert. Intel SGX applies abort page semantics; silently replacing the read data with a dummy value instead of raising a fault. Thus the race condition that enables Meltdown is absent when targeting Intel SGX. However, Intel SGX only applies the

abort page semantics after a legacy page table permission check succeeds without issuing a page fault. Foreshadow avoids these mechanisms by revoking all access permissions to the targeted enclave page, such that any access will lead to a page fault. This page fault leads to the race condition that enables the transient execution without abort semantics. Foreshadow then retrieves secrets in a similar vain as Meltdown; by measuring the time it takes to access certain cache lines. Furthermore, Foreshadow implements various optimization techniques to increase the bandwidth and reliability of secret extraction.

The SgxPectre attack can also extract secrets from Intel's architectural enclaves, but it exploits a different type of hardware vulnerability than Foreshadow [19]. Specifically, it presents ways to inject branch targets into SGX enclaves to trigger transient execution and leak secrets out of the enclave. This technique is similar to the Spectre attacks described earlier.

Lastly, the recent load value injection (LVI) attack presents new exploitations of Intel SGX that are not hindered by mitigations against Spectre and Meltdown [88]. LVI is not limited to hijacking branch outcomes, but allows to replace the result of any victim load micro operation with attacker-controlled data.

2.3 seL4 microkernel

seL4 is a microkernel of the L4 family, designed for applications in safety- and security-critical systems with high assurance and high performance [48].

As a microkernel, seL4 provides only a minimal set of features in kernel space. These features include inter-process communication (IPC), threads, virtual address spaces, capability-based access control, and interrupt control. seL4 leaves the implementation of other OS functionality — e.g., memory management, networking stack, device drivers — up to user space. This approach to kernel design creates a sufficiently small TCB that allows to formally prove its correctness.

As a microkernel, seL4 is unique in that it includes formal and automated proofs that the implementation is functionally correct, and that the formal model ensures confidentiality and integrity [47]. These proofs do not cover protection against timing channels — e.g., timing-based cache side-channels leveraged by Spectre and Meltdown (Section 2.2). However, the researchers behind seL4 are working towards formal proofs of timing-channel prevention [33, 41].

2.3.1 Capabilities

seL4 is a capability-based microkernel [71]. A capability is a reference to an object, which also encodes access rights to that object. For example, a read-write capability to a frame object gives the owner of the capability read and write access to a specific memory frame. Capabilities allow for delegation of access. *Minting* a capability copies it to a new capability that may have fewer rights. It can then be transferred to another entity. Revoking a capability strips the encoded access to the referenced object away from the holder of a capability, and recursively revokes any capabilities that were derived from it. Capabilities are monotonic, i.e., new capabilities will never have more access rights than the capability they are derived from.

An application's capabilities conceptually reside in its capability space (CSpace), which is implemented as a directed graph of capability nodes (CNodes). Every CNode is a table of slots, where each slot may contain other CNodes. In seL4, the kernel manages all capabilities and an application only references these by their addresses.

SeL4 bootstraps user space by assigning it a root capability to all available memory not used by the kernel. The initial user thread may split up the memory to create new, isolated applications. Thus, all new capabilities to subsets of the system's resources are derived from the root capability. The monotonic nature of capabilities ensures that no user space application has access to the kernel.

The capability-based access control also governs all seL4 kernel services. In order to perform any operation, an entity must invoke a capability with the respective access rights that it has in their possession. Capabilities always refer to a kernel object.

2.3.2 Kernel objects

The kernel objects defined by seL4 form the interface to the kernel itself. An application that interacts with the kernel, does so via the creation, manipulation and combination of the kernel objects. We only describe a subset of object types that are most relevant to this work.

Thread control blocks represent a thread of execution that may be scheduled, blocked, or unblocked depending on its interaction with other threads. A thread control block has an associated CSpace and virtual address space (VSpace).

Virtual address space objects construct a VSpace for one or more threads, effectively creating a sandbox around applications. A VSpace consists of a memory paging structure of at least one level, depending on the specific hardware-level architecture. For example, the following objects in seL4 correspond to the two-level paging structure

of the Arm 32-bit ISA.

1. `PageDirectory`
2. `PageTable`

The lowest level of the paging structure will be filled with `Frame` objects that correspond to frames of physical memory.

Endpoints facilitate message passing communication between threads, but also between applications and the kernel itself. IPC uses Endpoints and is synchronous. Capabilities to Endpoints can be restricted to send- or receive-only. Communication requires two Endpoints; one per direction. Capabilities can be sent in a message over Endpoints to transfer them.

Untyped memory objects are the foundational unit of memory allocation in seL4. Untyped objects can be *retyped* into other kernel objects. Additionally, they can be divided into smaller groups such that part of the system's memory can be delegated.

2.3.3 Memory allocation

The seL4 kernel does not dynamically allocate memory for kernel objects. Instead, applications must explicitly create objects by *retyping* Untyped kernel objects. The seL4 kernel provides system calls to retype Untyped objects. An application must have a sufficient capability to these Untyped objects in order to create new objects. These mechanisms for memory management enforce the isolation of physical memory between different applications.

Upon booting a system, the seL4 kernel sets up all memory. It first allocates the memory required for the kernel itself. seL4 then creates an initial user space thread to which it transfers all capabilities to the remaining Untyped memory, and a few capabilities to kernel objects necessary for bootstrapping the first thread. The kernel hands over control to the user space thread. The user space thread may split up the Untyped memory regions, create new threads or other kernel objects, and delegate authority to parts of the system as it sees fit.

2.3.4 CAmkES and capDL

To make practical development on top of seL4 easier, there is the CAmkES software library and framework. CAmkES supports component-based development of full software systems with strong isolation guarantees [52][73]. It provides interfaces between components via remote procedure calls (RPCs) for synchronous communication based on seL4 Endpoint kernel objects. The two other communication interfaces are *Events* for notifications and *Dataports* for shared data buffers. Each CAmkES component is al-

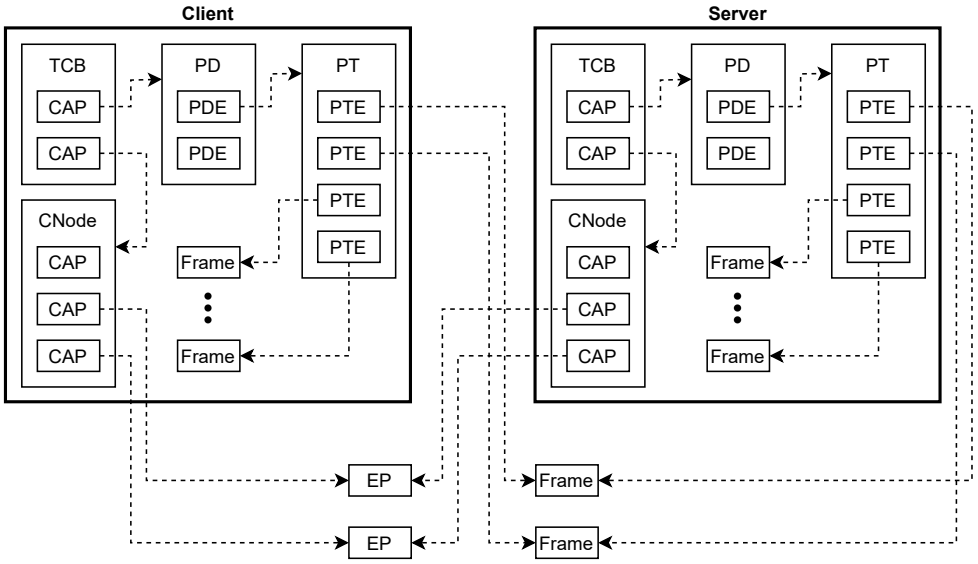

Figure 2.3. An example of an seL4 system and the capabilities and objects involved. There are two components: the Client and the Server. Their respective thread control blocks (TCBs) contain capabilities to the top levels of their VSpace and CSpace. Both components possess capabilities to two Endpoint objects (EP) used for synchronous inter-process communication (IPC). They also both map two Frame objects into their address space as shared memory for communication. The address space paging structure in this example is based on the implementation of Arm 32-bit ISA: page directory (PD), page table (PT), and their respective entries.

located a separate VSpace. This provides complete isolation between components, and only allows explicitly exposed, well-defined communication interfaces. A CAmkES application is defined using the architecture description language (ADL) that describes components, their assembly, connections, and configuration. The CAmkES framework processes all descriptions, combining it with code scaffolding, to build a complete and bootable seL4 system image.

The capability description language (capDL) provides a way to explicitly describe the capability distribution in seL4 systems [53, 74], i.e., a mapping between subjects and object capabilities. SeL4 can use a capDL specification during its bootstrapping process to simplify initialization of the system. CAmkES does this by default.

The seL4 project includes proofs that seL4 correctly initializes a system from boot state into a defined protection state described by a capDL specification [76].

A capDL specification consists of two sections: objects, and capabilities. The objects section describes all seL4 kernel objects in the system. The capabilities section specifies the capabilities that every component in the system is assigned. Figure 2.3 presents an example of a system with a client and a server component that communicate via two endpoints and two shared frames. Listings 2.4 and 2.5 contain the corresponding capDL specification.

```
1  objects {
2      client_tcb = tcb
3      client_cnode = cnode
4      client_pd = pd
5      client_pt = pt
6      frame_client_0000 = frame(4k, fill: [file, offset])
7      frame_client_0001 = frame(4k, fill: [file, offset])
8
9      server_tcb = tcb
10     server_cnode = cnode
11     server_pd = pd
12     server_pt = pt
13     frame_server_0000 = frame(4k, fill: [file, offset])
14     frame_server_0001 = frame(4k, fill: [file, offset])
15
16     example_endpoint_0 = ep
17     example_endpoint_1 = ep
18
19     shared_frame_0 = frame (4k, fill: [])
20     shared_frame_1 = frame (4k, fill: [])
21 }
```

Figure 2.4. The objects section of a capability description language (capDL) specification for the example system; simplified. The declaration of thread control block (TCB) objects on lines (2) and (9) will create the respective applications in the seL4 system. Subsequent lines declare the virtual address space (VSpace) and capability space (CSpace) objects. Both applications declare two frame objects (6-7) and (13-14) that are filled with their corresponding memory frames containing the application's executable. Lines (16-17) declare the Endpoint objects (EP) for communication between components. Lines (19-20) declare the shared Frame objects and keeps them empty.

```
1  caps {
2      client_tcb {
3          cspace: client_cnode
4          vspace: client_pd
5      }
6
7      client_cnode {
8          0x1: example_endpoint_0
9          0x2: example_endpoint_1
10     }
11
12     client_pd {
13         0x0: client_pt
14     }
15
16     client_pt {
17         0x0: frame_client_0000 (RWX)
18         0x10: frame_client_0001 (RWX)
19         0x20: shared_frame_0 (RWX)
20         0x30: shared_frame_1 (RWX)
21     }
22
23     -- server capabilities follow analogously --
24 }
```

Figure 2.5. The capabilities section of a capability description language (capDL) specification for the example system; simplified. Lines (2-5) initialize the thread control block (TCB) object with capabilities to the roots of the client's capability space (CSpace) and virtual address space (VSpace), which were defined in Listing 2.4. Lines (7-10) initialize the client's CSpace with capabilities to the two Endpoints (EP) in the system. Lines (17-18) map the application's executable in its VSpace with read-write-execute permissions. Lines (19-20) map the shared frames into the client's VSpace. The capability setup for the server component is similar to the client's, except that it uses its respective object names.

3. Problem statement

"The mystery of life isn't a problem to solve, but a reality to experience."

Frank Herbert, *Dune*

In this chapter we introduce the system and adversary models, in addition to the goals and requirements of our work. The system model provides a brief overview of the main components in the system and serves as context for the remainder of the chapter. The adversary model describes an adversary's goals and capabilities that in turn inform the requirements of a system. When evaluating a system, the achieved functionality and security are only meaningful in perspective of an adversary model. Finally, we formulate the goals and a set of concrete requirements to help evaluate our work.

3.1 System model

The system on which we build consists of three main components.

1. A rich execution environment (REE)
2. A trusted execution environment (TEE)
3. A co-processor

Figure 3.1 shows a simple system overview.

Firstly, the REE runs a general-purpose operating system (OS) — e.g. GNU/Linux — and any number of user space applications.

Secondly, the system has a separate TEE that runs a specialized, trusted OS. The system provides integrity guarantees between the REE and the TEE, such that the TEE can provide execution isolated from the REE. Applications from the REE can invoke functionality on the TEE and load their own trusted applications (TAs), but cannot directly manipulate memory or execution of the TEE.

Thirdly, the system contains a co-processor that is separated from the main processor. We assume there is a secure communication channel in place between the TEE and the co-processor. This co-processor performs sensitive cryptographic operations and stores cryptographic secrets. The manufacturer initializes the co-processor with a hardware root of trust (RoT).

The system has a hardware RoT that verifies the boot process of the firmware, which

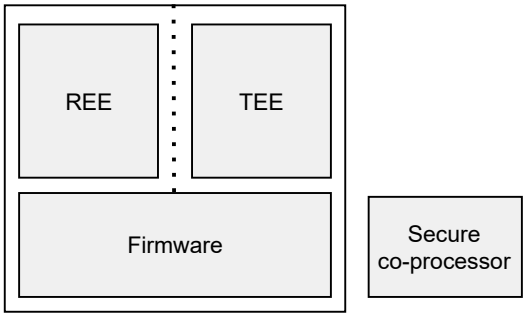

Figure 3.1. A minimal overview of the system model. The system is split into a separate rich execution environment (REE) and trusted execution environment (TEE). Execution and memory are isolated between the REE and TEE, though both still operate on the same system on chip (SoC). The secure co-processor is a separate chip for cryptographic operations that communicates with the TEE on the main SoC.

then ensures that the TEE is initialized correctly and separately from the REE. This is similar to the operation of Arm TrustZone (Section 2.1.3).

3.2 Adversary model

We consider an adversary with two goals: 1) forge attestation reports, and 2) execute unauthorized instructions in TAs. The adversary's first goal is to forge attestation reports to remote parties, in order to impersonate legitimate TAs. Consequently, the remote party might expose sensitive information when they continue communication with the adversary. The adversary's second goal is to compromise the integrity of a legitimate TA in order to alter its behavior.

We assume that the adversary controls the REE, including user space processes and the OS. Consequently, the adversary can load arbitrary TAs in our TEE. This means that the adversary can execute any malicious code in the TEE and attempt to forge TA-specific attestations. Lastly, we assume that the adversary is able to compromise confidentiality of the TEE's trusted OS and TAs. They would achieve this either by exploiting bugs in the trusted OS or TA code, or by leveraging side-channel attacks (Section 2.2).

However, we assume that the hardware is secure and uncompromised, and that the attacker cannot phsyically tamper with the hardware. Only a small fraction of real-world attackers will have the resources to tamper with the physical SoC of a target, thus we deem this a modest claim.

We assume that the secure co-processor guarantees confidentiality of its data and operations. This is a reasonable assumption, as there already exist secure elements that fit this claim — e.g., Apple's Secure Enclave [5], Google's Titan M [86], and the OpenTitan project[1].

[1] https://opentitan.org/

In summary, the assumptions that form our adversary model are as follows.

- The adversary has full control over the REE and can thus load and execute arbitrary TAs.
- The adversary can compromise confidentiality of TAs and the TEE OS.
- The adversary is unable to physically compromise the main processor or the co-processor.
- The adversary is unable to break confidentiality of the co-processor.

3.3 Goals and requirements

The motivation for this thesis work is to design an architecture for TEE remote attestation that guarantees integrity in the presence of an adversary as described in Section 3.2. The TEE must provide remote attestation of TAs, even when confidentiality of the TEE and TAs cannot be guaranteed. It must also guarantee integrity of other TAs and the TEE in the presence of a TA loaded and controlled by the adversary. Additionally, the TEE must support the secure dynamic loading of a new TA during run-time. We desire a relying party to be able to perform secure remote attestation and trust that the attested software is executed.

Our security requirements are thus as follows.

S-1: **Provide secure remote attestation even when confidentiality of the TEE and TAs cannot be guaranteed.**

S-2: **Retain isolation and integrity guarantees of TAs and the TEE, even in the presence of an adversary-controlled TA.**

S-3: **Support secure dynamic loading of TAs.**

Finally, we formulate a performance requirement for the attestation process of TAs.

P-1: **The performance for attestation of TAs should be affordable relative to the full execution cost of a TA.**

4. Design

"How do you cause people to believe in an imagined order such as Christianity, democracy or capitalism? First, you never admit that the order is imagined."

Yuval Noah Harari, *Sapiens*

4.1 High-level system design overview

Figure 4.1 shows the high-level system overview. Our design leverages the co-processor and the formally verified seL4 microkernel to achieve security properties neither provide on their own. We use seL4 as a basis for the trusted operating system (OS) of the trusted execution environment (TEE). We chose the seL4 microkernel because of its formally verified implementational correctness and formal model guaranteeing confidentiality and integrity (**S-2**). These properties provide strong isolation between all the components of our trusted OS design. The trusted OS manages the trusted applications (TAs) that will be executed in the TEE. Thus, the TAs will have the same isolation guarantees with respect to the TEE and other TAs.

However, seL4 does not formally prove protection against timing-based side-channel attacks. Though it provides mitigations for specific attacks — e.g., speculative execution attacks such as Spectre and Meltdown (Section 2.2.3) — it does not rule out the whole class of attacks. This is problematic when performing remote attestation, because it means that the confidentiality of cryptographic keys used to sign the attestation reports is not ensured. Previous work showed how the compromise of similar secrets lead to the breach of integrity in Intel's Software Guard Exentsions (SGX) enclaves that performed the attestation reporting (Section 2.2.3). An adversary can then forge attestation reports of TAs. To address this, our design relies on the secure co-processor to manage confidential data such as cryptographic secrets, and to perform sensitive cryptographic operations. This prevents the compromise of these cryptographic secrets and thus ensures integrity and confidentiality of the attestation procedure, addressing requirement **S-1**.

In our design we distinguish between a rich execution environment (REE) or non-secure world, and a TEE or secure world. This is derived from the split-world architecture of Arm TrustZone (Section 2.1.3).

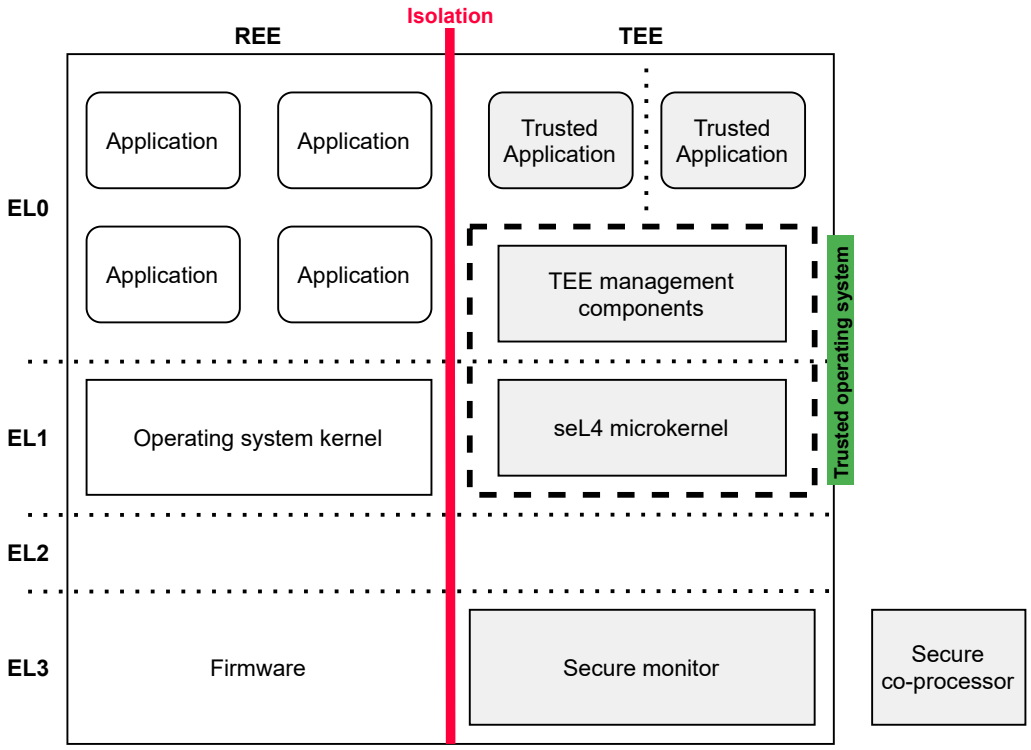

Figure 4.1. System architecture design with a rich execution environment (REE) and a trusted execution environment (TEE). The TEE runs a trusted operating system (OS) consisting of the seL4 microkernel and several management components. The trusted OS hosts every trusted application (TA) as an isolated component in itself. The confidentiality of the TEE maybe be compromised by side-channel attacks. Therefore, security critical data and operations are offloaded to the secure co-processor in the design.

4.2 Trusted OS design

The trusted OS of the TEE consists of the seL4 microkernel and multiple CAmkES components. Figure 4.2 shows an overview. There are four components that provide the TEE functionality: REEAPI, Loader, Attestation, and Driver.

REEAPI provides an application programming interface (API) to the REE for managing the TEE and TAs. It exposes three methods: 1) loading a TA, 2) attesting a TA, and 3) calling exposed TA functionality. To coordinate loading and attesting TAs, the REEAPI shares a remote procedure call (RPC) communication channel with the respective components implementing these functions in the TEE. However, the REEAPI cannot call functions exposed by a TA directly, because of seL4's strict isolation. All RPCs must be explicitly defined at build time. Thus, the REEAPI calls the Loader via a pre-defined RPC channel, with the function to be called as an argument.

Loader prepares a memory area for TAs to be loaded and initializes them into place. During run-time, the REEAPI may receive a request to load a specific TA into the TEE. Internally, it forwards this request to the Loader component. A TA is an executable and linkable format (ELF) binary file that is the product of a compilation process for

the TEE platform. The Loader receives a TA ELF file via a shared buffer with the REEAPI and correctly loads it into the prepared memory area such that its exposed functionality can be called, addressing requirement **S-3**. In the special case that a TA is already statically defined during the build process of the system, the Loader is not required to perform its normal operation anymore.

Attestation handles the requests for remote attestation of TAs. It first *measures* the TA that is to be attested. The measurement consists of computing a secure hash value over the TA's memory pages that contain its binary executable. This hash *fingerprint* uniquely identifies the state of the TA. However, in order to do this it must access the TA's memory, which is isolated from the Attestation component by default. Thus, we use seL4's capability system to assign the Attestation component the respective capabilities to the TA's memory pages. With read-only access, the Attestation component can execute its measurement.

Driver communicates with the co-processor on behalf of the TEE. For the remote attestation design, it receives the fingerprint that the Attestation component computed. The message to be signed may be extended to incorporate a nonce, timestamp, or other contextual information, besides only the TA fingerprint. The Driver composes a signing request message and sends it to the co-processor. We assume communication happens over a secure channel (Section 3.1), but we leave the specific design of this for future work. The Driver waits for a reply and subsequently returns the signature to the calling component.

4.2.1 Dynamic loading

We define dynamic loading as the provisioning and execution of a new TA in the TEE during run-time. Dynamic loading is a crucial feature in the scenario we are addressing with our work — i.e., TEEs that user-level code can leverage. When supporting dynamic loading, attesting the entire TEE image once during boot does not suffice. This therefore motivates our approach of attesting individual TAs.

The CAmkES framework for the seL4 microkernel requires all system components to be defined during the build process. The dynamic loading of new components is not supported, although this might change in the future [40]. Security requirement **S-3** states that the TEE must support dynamic loading of TAs while maintaining the secure remote attestation functionality (Section 3.3).

We formulate the process for dynamically loading TAs into the TEE during run-time as follows.

1. Developer statically compiles an application into an ELF binary file.

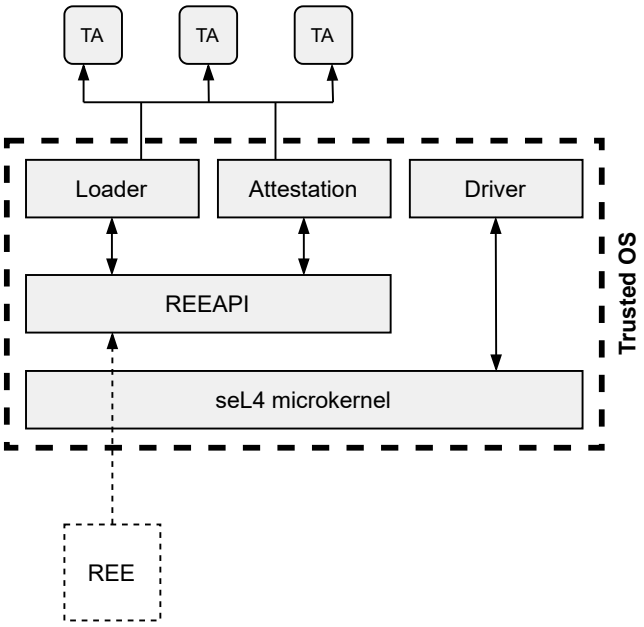

Figure 4.2. Overview of the components in the trusted operating system (OS). The trusted OS is built with the seL4 microkernel, that provides strong isolation between different components. The REEAPI handles any calls coming into the trusted execution environment (TEE) from the rich execution environment (REE). The Loader initializes a trusted application (TA) in memory. The Attestation component computes a hash fingerprint of a TA and returns an attestation report. The Driver communicates with the co-processor and requests the cryptographic signing operations.

2. REE application uses the application binary interface (ABI) to load the binary ELF file into the TEE.
3. Loader parses the ELF file.
4. Loader loads the TA sections/segments into the *Abstract TA* component.
5. Abstract TA stores a symbol table to call the exported functions of the TA.

An Abstract TA is a CAmkES component, similar to any of the other components in the TEE. It serves as a container for actual TAs that are loaded into the TEE. In this manner, any dynamically loaded TA is still contained in a properly isolated component enforced by seL4's capability-based access control.

The Abstract TA shares a memory buffer with the Loader component to hold a TA executable. The Attestation component needs access to the same memory buffer to compute the fingerprint. Due to limitations in CAmkES, a memory buffer cannot be shared between more than two components. Therefore, we merge the Loader and Attestation components and share the buffer in the Abstract TA with this new component. This component will perform the dynamic loading and compute the fingerprint for attestation.

The concept of the Abstract TA is still constrained by the static system definition enforced by seL4 CAmkES. This means that an Abstract TA component has to be statically defined during the build process of the system. The number of frame objects

it has allocated is fixed, resulting in a maximum supported binary size for TAs. The choice of this size limit is an implementation decision that is informed by physical platform constraints and assumptions on the TAs.

As an additional consequence, the number of Abstract TA components in the system is fixed, as they cannot be created dynamically during run-time. Therefore, creating one Abstract TA for every TA is infeasible, assuming the TEE has no prior knowledge of how many TAs will be loaded. An implementation must thus decide on a fixed number of Abstract TA components to define. When the number of TAs surpasses the available Abstract TA components, the Loader must replace the contents of one Abstract TA to load a new TA. Further management of TA state in those cases is left for future work.

4.3 Secure co-processor

The TEE and the co-processor communicate in a request-response message exchange pattern. The TEE sends a message to request certain operations from the co-processor, and the co-processor responds to that request in a new message.

The original equipment manufacturer (OEM) provisions the co-processor with its cryptographic keys as described in Section 4.4. In our design, we use the Ed25519 public-key signature system [10] on the co-processor. Importantly, Ed25519 is a high-security signature scheme. Additionally, its signing operations are fast and the size of the keys and signatures is small. These are valuable properties for employment on a naturally more resource-constrained co-processor.

4.4 Remote attestation protocol

Figure 4.3 illustrates the attestation flow. On a high level, remote attestation in our system design works as follows.

0. Loader dynamically loads TA α into memory.
1. REEAPI receives a request for attestation of TA α and forwards it to Attestation.
2. Attestation computes a hash of TA α's code and data sections in memory.
3. Attestation asks the Driver to request the co-processor to sign the hash value. The co-processor signs the hash using its attestation key and returns it.
4. The trusted OS creates an attestation report for TA α and sends it to the requesting party.

The provisioning of cryptographic credentials to the co-processor happens during manufacturing. The OEM issues device key certificates and publishes these in an

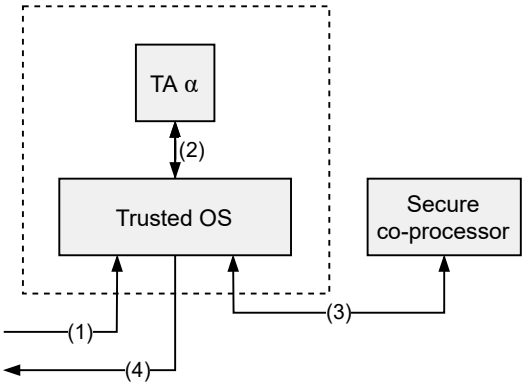

Figure 4.3. The remote attestation flow in our system design. The trusted application (TA) α is already loaded. The trusted operating system (OS) receives a request for attestation in 1) and performs the attestation of trusted application (TA) α in 2). It then sends the fingerprint to the co-processor to be signed in 3) and returns an attestation report to the relying party in 4).

OEM database. Later, a relying party can request these to verify that the signing key used in the signing operations belongs to a legitimate co-processor.

The described remote attestation protocol serves as a minimal proof-of-concept (PoC). In future work we envision a more robust attestation protocol, including freshness guarantees, a key hierarchy, and a public key infrastructure (PKI). Our design is fundamentally about ensuring integrity of the cryptographic keys.

5. Implementation

"It feels like trying to juggle eight-side Rubik's cubes while trying to solve them at the same time. And every time I drop one, God kills a billion kittens."

Hannu Rajaniemi, *The Fractal Prince*

5.1 Trusted OS

We developed a proof-of-concept (PoC) of the trusted operating system (OS) using the seL4 microkernel and the CAmkES application framework in the C programming language. The trusted OS has several components that together provide the trusted execution environment (TEE) functionality. Trusted applications (TAs) are executed as distinct CAmkES components on top of the trusted OS.

The complete source code repository is publicly available [1].

5.1.1 Execution environment and build process

The PoC runs in an emulated 64-bit Arm virtual machine, on an Intel Core i5-10310U host system running Ubuntu 20.04 LTS. We use QEMU 4.2 [2] as our machine emulator. To allow communication between the trusted OS and the co-processor, we configure input and output pipes that we pass through QEMU as a serial device to seL4. We develop a script local to the host machine that forwards data between the configured pipes, thereby acting as a bridge from the emulated trusted OS to the co-processor that is connected to the host machine.

The build process of the trusted OS is based on CMake [3]. Source dependency management for seL4 projects is done using Repo [4]. To use Repo, a project must have an XML manifest file to describe its structure and dependencies. We extend the CAmkES manifest file [72] to point to our project-specific repositories. Furthermore, the seL4 foundation provides build tools for seL4 and CAmkES as a set of Dockerfiles [77]. Running these gives an environment in which to compile the seL4 project and our CAmkES application.

[1] https://github.com/ssg-research/sel4-tee
[2] https://www.qemu.org/
[3] https://cmake.org/
[4] https://gerrit.googlesource.com/git-repo/

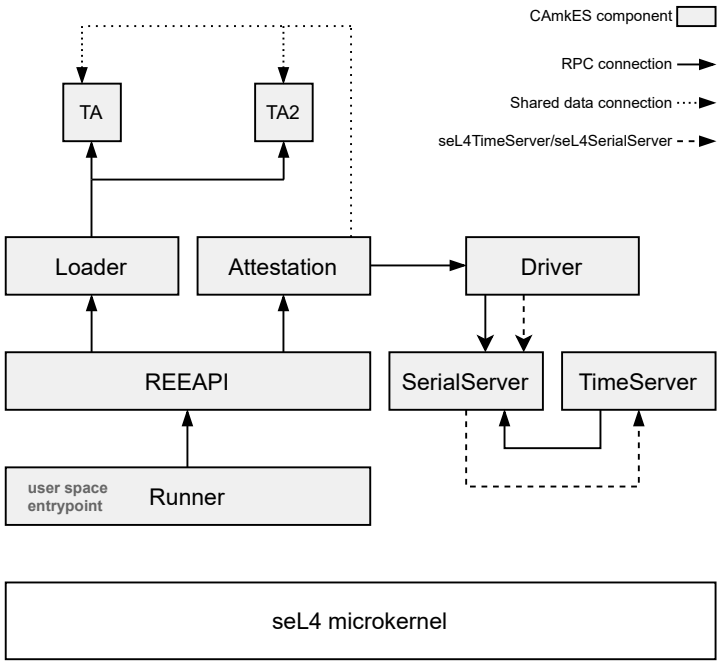

Figure 5.1. System overview of the trusted operating system (OS) implementation in seL4 and CAmkES.

```
|-- CMakeLists.txt /* CMake build automation configuration  */
|-- components /* directory for all TEE management CAmkES components */
|   |-- Attestation
|       |-- Attestation.camkes /* component-specific CAmkES configuration file */
|       |-- CMakeLists.txt /* component-specific build configuration */
|       |-- src /* source code of the component */
|           |-- attestation.c
|   ...
|-- interfaces /* directory for all function interface definitions */
|   |-- Attest.idl4
|   ...
|-- tee.camkes /* CAmkES application configuration */
```

Listing 5.1. An abbreviated file tree of the proof-of-concept (PoC) trusted execution environment (TEE) implementation using CAmkES.

5.1.2 TEE management components

The PoC comprises several CAmkES components that implement subsets of the TEE functionality: REEAPI, Loader, Attestation, Driver, and Runner (Section 4.2). Figure 5.1 shows a system overview of the PoC implementation. The Runner is specific to the PoC implementation and serves as a way to emulate a rich execution environment (REE) application that interacts with the TEE. It invokes the REEAPI via remote procedure calls (RPCs), that use seL4 Endpoint kernel objects internally (Section 2.3.2). A real-world REE application would invoke these functions similarly, but then via the application binary interface (ABI) (Section 4.2.1).

The project layout is split into several CAmkES configuration files for components, interfaces, and the application itself. Listing 5.1 shows an overview.

```
1  import "../../interfaces/Attest.idl4";
2  import "../../interfaces/Offload.idl4";
3
4  #include <buffer.h>
5
6  component Attestation {
7      provides Attest attest;
8      uses Offload offload;
9      dataport Buf(TA_BUFSIZE) ta;
10     dataport Buf(TA_BUFSIZE) ta2;
11 }
```

Listing 5.2. Example Attestation component definition in CAmkES. Line 1-2 load the required interfaces. Line 4 includes definitions for the configured buffer size. Line 7 registers the interface this component provides. Line 8 registers that this component uses the Offload interface. Lines 9-10 define shared memory buffers with every trusted application (TA).

```
1  import "components/Attestation/Attestation.camkes";
2  import "components/REEAPI/REEAPI.camkes";
3
4  assembly {
5      composition {
6          component Attestation att;
7          component REEAPI api;
8
9          connection seL4RPCCall api_attest_conn(from api.attest, to att.attest);
10     }
11 }
```

Listing 5.3. Top-level CAmkES configuration to connect the REEAPI and Attestation components using a remote procedure call (RPC) connection.

Components and their connections are described in CAmkES configuration files using the architecture description language (ADL). We define the Attestation component as shown in Listing 5.2. Listing 5.3 contains top-level configuration to connect two components using a RPC. The CAmkES build system uses these definitions to generate *glue-code* that sets up the described system using seL4 primitives during system boot. In this example, both the REEAPI and Attestation component will be initialized with their own thread control block and associated virtual address space (VSpace) and capability space (CSpace) (Section 2.3). CAmkES then connects them via seL4 Endpoint objects, that transparently pass the function invocation and return values.

CAmkES calls these RPC communication channels *procedures*. The functional interface of a procedure is defined in a separate interface description language (IDL) file. Listing 5.4 shows the Offload procedure we develop as the functional interface for the Driver component.

```
1  procedure Offload {
2      int sign(in char fingerprint[], out char signature[]);
3  }
```

Listing 5.4. The interface definition for the Offload procedure provided by the Driver component. It exposes a ‘sign‘ function that takes a fingerprint and outputs a signature.

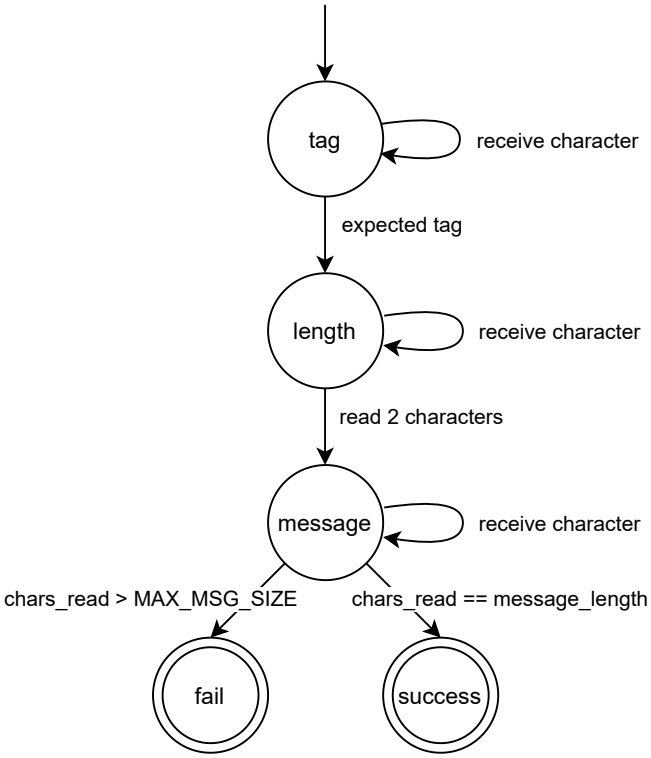

Figure 5.2. A state diagram describing the serial response handler in the Driver component.

We write the TEE management component source files in C. We implement every component in accordance with the function signature from the interface definition. Function names are prefixed with the component-local name of the interface and an underscore, e.g. `offload_sign`.

The Driver component implements the communication protocol with the co-processor (Section 5.3). It reuses the CAmkES `SerialServer` global component [75] that abstracts over the platform's serial device for communication. The Driver waits for a response from the co-processor, reading it character-by-character when it is sent, following the state machine as described in Figure 5.2.

5.1.3 Trusted applications

In our PoC TEE, we define and execute TAs as CAmkES components. This means that they must be statically defined in the trusted OS (Section 2.3.4). We refer to every statically loaded TA using a unique identifier, as shown in Listing 5.5. Based on this identifier, the Loader component invokes the registered entrypoint of a TA via a RPC. The lifecycle of a TA in this scenario comprises three stages: registering, loading, and executing.

TA developers must register their applications in the TEE. Registration of TAs in our PoC involves the following steps.

1. Create the TA component source in the `components` directory (Listing 5.1).

```
// include/trusted_apps.c
enum tee_registered_ta
{
    ID_TA1 = 1,
    ID_TA2 = 2,
};
```

Listing 5.5. The C header file specifying unique identifiers for every statically loaded trusted application (TA).

```
assembly {
    configuration {
        att.ta_access = "R";
        ta.d_access = "R";
    }
}
```

Listing 5.6. The C header file specifying unique identifiers for every statically loaded trusted application (TA).

2. Define the component in the composition block in `tee.camkes` (Listing 5.3).
3. Register a unique identifier for the TA in the `include/trusted_apps.h` header file (Listing 5.5).
4. Register lowercase name of the TA component in the TEE mapping scripts `remap_dataports.py` and `resize_dataports.py` (Section 5.1.4).
5. Write the functional interface that the TA provides (Listing 5.4).
6. Register the entrypoint of the TA by invoking it from the Loader.
7. Set up a shared memory buffer between the TA and Attestation components.
    a. Declare CAmkES dataports in both components of default type `Buf` with size set to `{TA NAME}_BUFSIZE` (Listing 5.7).
    b. Define shared connection between both dataports in the composition block in `tee.camkes` (Listing 5.8).
    c. Configure access to be read-only (Listing 5.6).

The loading of TAs occurs after the system boots, as part of the user space initialization. To start a TA, the Runner calls the REEAPI and passes a specific TA identifier. The REEAPI calls the Loader, which invokes the entry point of the respective TA, thereby handing over the thread of control to the TA.

5.1.4 Attestation measurement

By default, seL4 strongly isolates all components by virtue of its capability-based access-control system. However, we require the Attestation component to read the memory of TAs in order to perform remote attestation. Our PoC uses CAmkES dataports to share the TA's memory with the Attestation component. We map the memory frame objects containing the TA binary executable into the virtual address space of the Attestation component.

```
import "../../interfaces/TAInterface.idl4";
#include <buffer.h>

component TA {
    provides TAInterface app;
    dataport Buf(TA_BUFSIZE) d;
}
```

Listing 5.7. Trusted application (TA) configuration with dataport definition.

```
assembly {
    composition {
        component Attestation att;
        component TA ta;
        connection seL4SharedData ta_binary(from ta.d, to attestation.ta);
    }
}
```

Listing 5.8. Shared data connection between a trusted application (TA) and the Attestation component.

As a prerequisite for statically loaded TAs, there must be a shared data connection with the Attestation component. Both the Attestation and the TA declare a unique dataport, as shown in Listing 5.7. We then connect these two dataports in the top-level CAmkES configuration (Listing 5.8).

There are two challenges when implementing this that both stem from a circular dependency. First, we need to know the size of the TA to reserve a sufficiently large memory area for the Attestation's dataport. However, the size of the TA is only known after compilation. Second, we require the memory frame objects that contain the TA executable to map into the Attestation's dataport, but those are again only known after compilation. We write two scripts to solve these problems. They execute in between multiple rounds of compilation, such that they solve the circular dependency.

Listing 5.9 shows part of the `resize_dataports.py` script that determines the executable size of a TA and resizes the dataport in the Attestation component. To determine the size of the compiled TA, the script counts the number of pages in the capability description language (capDL) specification that are filled with file data. The objects section of the capDL specification specifies all frame objects for the TA, but only a subset of those are initially filled (Section 2.3.4). The default page size on the Arm platform is 4096 bytes. Thus, the size of the compiled TA is $4096 \times ta\_code\_page\_count$ bytes. The script stores this as a TA-specific C preprocessor macro in the dedicated `buffer.h` header file, e.g. `#define TA_BUFSIZE 40960`. By triggering a new compilation, the Attestation component now initializes the dataport to the size of the compiled TA.

The next challenge is to provide the Attestation component with the correct capabilities to access the TA's executable code and data. The idea is to map those frame

```
for ta_name in ta_names:
    keyword_frame = f'{ta_name}_group_bin'
    keyword_code = f'CDL_FrameFill_FileData "{keyword_frame}"'

    ta_code_page_count = len([i for i in findall(keyword_code, spec)])

    print(':: Found {} executable pages for TA "{}"'.format(ta_code_page_count, ta_name))

    try:
        tmp = enumerate(lines)
        index = [i for i, s in tmp if '{}_BUFSIZE'.format(ta_name.upper()) in s][0]
    except:
        lines.append('')
        index = -1

    new_define = '#define {}_BUFSIZE {}\n'.format(ta_name.upper(), 4096*ta_code_page_count)
    lines[index] = new_define

open(args.header, 'w').writelines(lines)
```

Listing 5.9. Shortened version of `resize_dataports.py` that determines the executable size of every trusted application (TA) and dynamically resizes the respective dataport in the Attestation component.

objects into the virtual memory area initialized for the shared buffer in the Attestation component's VSpace. The CAmkES programming interface does not provide access to the seL4 primitives that allow to assign capabilities. Instead, our second script `remap_dataports.py` (Listing 5.10 shows a shortened version) manipulates an intermediate capDL representation of the capability distribution in our TEE to achieve a correct mapping. The scripts first retrieves all indices of the frame objects containing the TA binary executable by iterating over the system objects that are specified as shown in Listing 5.11. Then, it finds the indices of the frame objects associated with the shared data connection. Finally, the script finds the object that assigns the frame object capabilities into the CSpace of the Attestation component. It inserts the indices corresponding to the TA's binary executable frame objects into the `.obj_id` field of the respective capability slots (Listing 5.12). A new round of compilation now sets up the complete capability distribution such that the Attestation component has direct access to the TA's loaded executable from its own virtual memory address space.

Calculating the hash fingerprint is straightforward. The Attestation component includes TweetNaCl [11] for the cryptographic implementations. It applies the SHA-512 hashing operation to the memory area that contains the TA's executable. The Attestation components invokes a RPC at the Driver to offload the signing operation. For observability purposes of our PoC, we return the resultant signature as a Base64-encoded string.

```
for ta_name in ta_names:
    connection_name = '{}_binary'.format(ta_name)

    # 1. Find all code frames of the TA
    # a. First retrieve all frames of the TA, later prune out irrelevant frames
    nr_ta_frames = spec.count('.name = "frame_{}_group_bin'.format(ta_name))
    offset = 0
    frame_indices = []
    for _ in range(nr_ta_frames):
        i = spec.find('.name = "frame_{}_group_bin'.format(ta_name), offset)
        frame_indices.append(i)
        offset = i + 1
    # b. Loop over frame_indices and save indices of the ones with FileData
    filedata_frame_indices = []
    for i in frame_indices:
        post = spec[i:i+200]
        if post.find('CDL_FrameFill_FileData') != -1:
            filedata_frame_indices.append(i)
    # c. Get object IDs of the code frames
    code_obj_ids = []
    for i in filedata_frame_indices:
        pre = spec[i-40:i]
        obj_id = pre[pre.find('[')+1:pre.find(']')]
        code_obj_ids.append(obj_id)

    # 2. Find dataport Frame Objects
    count_dataport_frames = spec.count('.name = "{}_data'.format(connection_name))
    dataport_obj_ids = []
    for i in range(count_dataport_frames):
        offset = spec.find('.name = "{}_data_{}_obj'.format(connection_name, i))
        pre = spec[offset-40:offset]
        obj_id = pre[pre.find('[')+1:pre.find(']')]
        dataport_obj_ids.append(obj_id)

    # 3. Find cap for dataport in the Attestation component
    # a. First find index of start of Attestation capability group
    index_attest_caps = spec.find('.name = "pt_attestation_group_bin')
    # b. Find Attestation's FrameCaps for dataports
    tmp = findall('/* {}_data'.format(connection_name), spec[index_attest_caps:])
    framecaps = [index_attest_caps + i for i in tmp]
    # Select only those that are part of Attestation capabilities
    framecaps = framecaps[0:count_dataport_frames]

    # 4. Insert obj_id for code frame into dataport caps
    for i in range(len(dataport_obj_ids)):
        area = spec[framecaps[i]-16:framecaps[i]]
        area = area.replace(dataport_obj_ids[i], code_obj_ids[i])
        spec = spec[:framecaps[i]-16] + area + spec[framecaps[i]:]
```

Listing 5.10. Shortened version of `remap_dataports.py` that finds the specific frame objects containing a trusted application (TA) executable and map those into the respective Attestation dataport.

```
[2036] = {
#ifdef CONFIG_DEBUG_BUILD
.name = "frame_ta_group_bin_0002",
#endif
.type = CDL_Frame,
.size_bits = 12,
.frame_extra = { .paddr = 0,.fill = { {.type = CDL_FrameFill_FileData,
.dest_offset = 0,
.dest_len = 4096,
.file_data_type = {.filename = "ta_group_bin",
.file_offset = 8192
}},}
 },
},
```

Listing 5.11. Capability description language (capDL) C specification for trusted application (TA) frame objects.

```
[2449] = {
#ifdef CONFIG_DEBUG_BUILD
.name = "pt_attestation_group_bin_0002",
#endif
.type = CDL_PT,
.slots.num = 429,
.slots.slot = (CDL_CapSlot[]) {
...
{350, {.type = CDL_FrameCap, .obj_id = 2036 /* ta_binary_data_1_obj */,
    .is_orig = true, .rights = (CDL_CanRead), .vm_attribs = seL4_ARCH_Default_VMAttributes,
    .mapping_container_id = INVALID_OBJ_ID, .mapping_slot = 0}},
...
}
```

Listing 5.12. Mapping frame objects into the capability slots of the Attestation component in the capability description language (capDL) C specification.

Tag	Description	Data
RR	Request public key	-
RP	Response public key	Ed25519 public key
RM	Request signature	Bytes
RS	Response signature	Ed25519 signature

Table 5.1. Message types for communication between the trusted execution environment (TEE) and co-processor. These messages are encoded in a tag-length-value (TLV) scheme.

5.2 Co-processor

We [5] implement the co-processor using a Digilent Arty A7-100T field-programmable gate array (FPGA) development board that runs a SiFive Freedom E300 RISC-V processor. It exposes two interfaces for communication with a host machine: a serial connection and a debugging connection. The universal asynchronous receiver/transmitter (UART) interface provides an RS-232 interface for communication over a serial connection. In our case, this happens over the USB-to-serial interface that is present on the Arty A7-100T development board. Additionally, it exposes a joint test action group (JTAG) interface to program and debug the softcore of the development board. We use this connection to flash the RISC-V softcore.

We provision the co-processor with software that continuously waits for messages on the serial connection, identified by a special prefixed character: the ASCII data-link escape (DL; `0x10`). Its cryptographic operations are implemented by the TweetNaCl [11] library. In the PoC, our bridge application on the host provisions the co-processor with its key pair. In a real-world setting, this is done by an original equipment manufacturer (OEM) during the manufacturing process. The co-processor uses this key pair to perform the cryptographic signature operation using TweetNaCl's implementation of the Ed25519 scheme.

5.3 Communication protocol

The trusted OS and the co-processor communicate via a single serial communication channel. We encode messages in a tag-length-value (TLV) scheme, as illustrated in Figure 5.3. Every message starts with a fixed size *tag* field that identifies the type of the message. The next field encodes the length of the message contents. The value or content of the message is in the last field, which is of dynamic size dependent on the length of the value. We describe four message types in Table 5.1: a request and response pair for the public key of the co-processor, and a request and response pair for the signing operation.

[5] The implementation of the co-processor was not done by the author of this thesis, but instead by another researcher in the group.

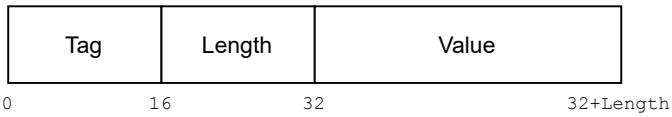

Figure 5.3. The tag-length-value (TLV) encoding scheme for messages between the trusted execution environment (TEE) and the co-processor. The first sixteen bits contain the tag that identifies the *type* of the message. The bits 16-31 encode the length of the value field containing the data of the message. The data field can thus be of arbitrary length.

5.4 Dynamic loading of ELF files

Our PoC TEE does not implement the dynamic loading of TAs. This section instead discusses some considerations for future implementations.

The TA executable and linkable format (ELF) blob must be sent to seL4 via the ABI, and within seL4 the binary must be passed between components via dataport connections: from REEAPI, to Loader, into Abstract TA.

To dynamically load a new TA, the Loader must copy the ELF blob into the reserved memory area of the Abstract TA. However, there are additional considerations to take into account. In order to address specific functions in the TA, the Loader has to retain the symbol table or parse it again, provided that it is not stripped. Given a function name, it must find the entry in the symbol table and associated string table to determine the offset in the `.text` section for the respective function.

Additionally, as the Abstract TA component exists before the TA is loaded, there is little influence over the virtual memory address space. Thus, a TA must be compiled as position-independent code (PIC).

Dynamic executables require dynamic linking of libraries at run-time. However, to do this from seL4 user space is more complicated, because we cannot easily map additional libraries into the VSpace. By default, the ELF files built by seL4 are statically linked and have no relocations. SeL4's ELF loader tool copies these blobs into memory without any additional linking [6] A minimal implementation of dynamic loading would support only these types of TAs. The loading of an ELF blob might then encompass the following steps.

- Pre-validations (memory bounds, page alignment, file format)
- Zero out target memory region, because the ELF file may be sparse
- Load each segment of the ELF file
  - Skip segments not marked as loadable
  - Parse program header table
  - `memcpy` data

[6] https://github.com/seL4/seL4_tools/blob/master/elfloader-tool/src/common.c

6. Evaluation

"Most people are not naturally reflective any more than they are naturally malicious."

James Baldwin, *Stranger in the Village*

We evaluate our trusted execution environment (TEE) against the requirements formulated in Section 3.3. The TEE must provide remote attestation that is secure in the presence of side-channel attacks that compromise confidentiality of the TEE. It must also continue to provide isolation and execution integrity for components of the TEE and trusted applications (TAs). Finally, the performance overhead of the remote attestation procedure must be affordable relative to the full execution cost of a TA.

6.1 Security analysis

The TEE must do secure remote attestation and provide guarantees that the attested TA is also the software that is executed (Section 3.3). This requires integrity and confidentiality protection of the attestation procedure, and isolation properties for components in the TEE. We evaluate these two requirements separately.

6.1.1 Secure remote attestation (S-1)

Our TEE aims to protect the integrity and confidentiality of remote attestation via the introduction of a secure co-processor. For it to fulfill requirement **S-1**, the TEE must ensure that the attestation keys and attestation operations are not accessible to an adversary, either physically, via side-channels, or using other software attacks.

The TEE must be correctly initialized. Our system model assumes that a hardware root of trust (RoT) verifies the boot process and initialization of the TEE. This ensures that the TEE is in a trusted state for remote attestation.

As per our system and adversary models, the co-processor is secure from adversaries (Sections 3.1 and 3.2). Specifically, it is resistant against the side-channel attacks discussed in Section 2.2, because the co-processor and the main processor do not share any caches or other hardware units that proved to be exploitable. Additionally, our model assumes that an adversary cannot physically access the co-processor. Finally, the co-processor is a fixed-function device in our system. Consequently, it does not allow

the execution of any arbitrary code, limiting the vulnerability to software attacks.

Furthermore, only the co-processor stores the secret cryptographic keys used for attestation. These keys are never sent to the TEE. All sensitive cryptographic operations — i.e., Ed25519 signing operations — that the relying party will verify are also only performed on the co-processor.

An adversary must not be able to change the hash fingerprint before it is attested. In our system, the fingerprint never leaves the trusted computing base (TCB) before it is attested, as illustrated in Figure 6.1. The TCB comprises the components that have critical control over parts of the attestation procedure, that would allow them to compromise integrity. The REEAPI is not strictly part of the TCB because it cannot interfere with the attestation procedure, except for refusing to invoke the Attestation component. However, it can modify the TA ELF binary as it flows through the system during dynamic loading. Even though this is strictly separate from the attestation procedure, we include the REEAPI in the TCB anyway because it is capable of interfering with the process as a whole.

The Attestation component computes the hash fingerprint by reading the TA's memory directly, and sends it to the co-processor via the secure communication channel. The co-processor computes a signature and returns it to the TEE, which forms an attestation report that it eventually sends back to the relying party. An adversary thus cannot violate the fingerprint's integrity.

As a consequence of these properties, our TEE maintains the remote attestation security. These security guarantees continue to hold even in the presence of the microarchitectural side-channel attacks we presented.

The focus of our work was not to design any specific attestation protocols. Hence, the proof-of-concept (PoC) TEE only implements a placeholder protocol. A sound attestation protocol would additionally require temporal consistency — e.g., a nonce or another proof of freshness (Section 2.1.1).

6.1.2 Isolation and integrity of the TEE and TAs (S-2)

A TEE guarantees users that their TA executes in a safe environment, where its execution cannot be tampered with by other TAs or even from outside the TEE. In order to maintain these guarantees, a TEE must provide strong isolation and integrity protection. This additionally affects requirement **S-1**. A user can still not trust a system that correctly loads their application, if it does so in an unsafe environment. To fully provide secure remote attestation, our TEE must also ensure that the attested code is actually the code that executes.

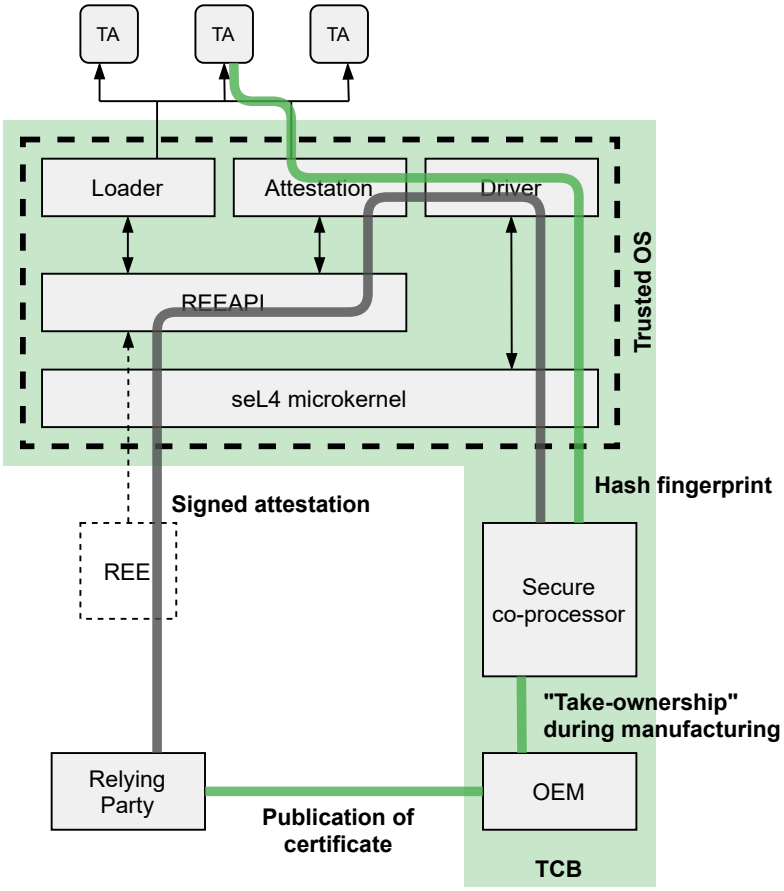

Figure 6.1. Graphical security analysis of the attestation flow based on the work in [38]. The Attestation component computes the hash fingerprint by directly accessing the memory of the trusted application (TA). The fingerprint never leaves the trusted computing base (TCB) while it is sent to the secure co-processor for signing (green line). The co-processor is provisioned with its cryptographic keys during manufacturing. Only the signed fingerprint ever crosses the TCB boundary (gray line), when it is sent to the relying party. The relying party verifies the signature using public certificates published by the original equipment manufacturer (OEM).

The seL4 microkernel uses capability-based access-control (Section 2.3.1). Such a capability-based system is instrumental for strong isolation. A capability must be granted explicitly to any component before the component can operate on an object. All components in our TEE are CAmkES components (Section 4.2), including the TAs. The system creates every CAmkES component by allocating it capabilities to a subset of the system objects. These are only used for that component. The only exceptions are objects related to inter-process communication (IPC), and buffers used by the Loader and Attestation components (Section 5.1.2). The IPC objects are contained in the seL4 proofs, which gives a convincing basis for trusting their integrity. In the static loading PoC, the Loader does not have any access to TAs. We discuss the case of dynamic loading later. The Attestation component has capabilities to the Frame objects of all TAs. However, these capabilities are read-only. In conclusion, the isolation and integrity of the TEE and TAs are guaranteed, and the defined IPC channels and shared objects do not compromise these guarantees.

6.1.3 Secure dynamic loading (S-3)

The dynamic loading of TAs raises additional issues compared to static loading. Firstly, the TEE must make sure that the TA code is not modified after loading and attestation. Our TEE is built on top of the formally verified seL4 microkernel that ensures strong isolation (Section 2.3). Furthermore, our dynamic loading design proposes an Abstract TA CAmkES component (Section 4.2.1). The Abstract TA component acts as a container for a dynamically loaded TA, ensuring that it adheres to the same isolation and integrity guarantees as any other component in the system. This means that a dynamically loaded TA is not able to interfere with the rest of the trusted operating system (OS), and neither are other components able to compromise the integrity of the TA. The exceptions to this are the explicitly defined communication channels with the merged Loader and Attestation component. In our design, the Loader part of the merged component needs write-capabilities to the memory area from where the TA is executed. The Attestation part needs capabilities to the same memory area, but these are read-only, and they therefore do not influence integrity. Our system model assumes that both the Loader and Attestation are part of the TCB — as shown in Figure 6.1. Merging these components does therefore not affect security. An adversary can also not leverage the write-capabilities of the Loader part to violate integrity of a TA. In conclusion, this ensures that a dynamically loaded TA cannot be modified by an adversary after attestation.

Secondly, the TEE must ensure that the correct TA is launched. Our PoC does not implement specific features that address this concern, though it addresses distinct static TAs using a unique identifier. In the case where there is only one Abstract TA, execution of the correct TA follows logically after the dynamic loading operation. When there are more Abstract TA components, the TEE must keep track of the state of every TA. Future work can extend the design to address these challenges.

6.2 Performance considerations

The remote attestation procedure incurs performance overhead from two sources: measuring TAs and signing measurement fingerprints.

Chapter 5 described the PoC TEE. The Attestation component in the seL4-based trusted OS computes the hash fingerprints using the SHA-512 hash function. The co-processor signs the fingerprints using the Ed25519 signature scheme.

The system model assumes no confidentiality for the environment in which the attestation measurement is performed (Section 3.1). Therefore, our measurement procedure

can use any efficient hashing algorithm. Specifically, the measurement fingerprint is a SHA-512 hash computed by the Attestation component. The SHA-2 family — which includes SHA-512 — is widely used in applications and protocols, and there exist many optimized implementations, both in software and in hardware [18]. Furthermore, the attestation measurement is only performed once during the lifecycle of a TA. The remote attestation our TEE provides only applies to the static state of a TA after it is loaded, but before it is executed. During the rest of the TA's execution, there is no performance overhead as a consequence of attestation. Additionally, performance overhead will also be minimal considering that TAs are generally single-purpose applications and thus smaller in size than regular applications in the rich execution environment (REE). Combining these arguments, we consider the performance overhead from the measurement of TAs to be small.

The co-processor signs the measurement fingerprint it receives using the Ed25519 signature scheme. Ed25519 provides very fast signing operations [10] and there is also recent work on fast and compact hardware implementations of the signature scheme [87]. Our co-processor implementation (Section 4.3) is however not representative for performing concrete performance measurements of the attestation operation, because it is not integrated with the system on chip (SoC) similar to real-world implementations [5, 86]. However, the fingerprints that the co-processor will sign are of a fixed, small size. Thus, there is little overhead when computing the signature of the fingerprint for attestation.

We argued that the two sources of performance overhead for remote attestation in our TEE are small, thus fulfilling requirement **P-1**.

7. Discussion

"But it is one thing to read about dragons and another to meet them."

Ursula K. Le Guin, *A Wizard of Earthsea*

In our proof-of-concept (PoC) trusted execution environment (TEE), we use the CAmkES application framework for seL4. This makes development of applications easier, but it also limits what we are able to do. For example, using the CAmkES framework, every component must be statically defined during development. CAmkES abstracts over the manual capability manipulation functionality provided by seL4, making them inaccessible from a component's perspective. Instead, we could have developed our TEE without CAmkES. This would give us greater control over the capability distibution during run-time. Consequently, it allows to copy capabilities to a shared memory region to multiple components — i.e., a trusted application (TA), Loader, and Attestation component — without CAmkES's dataport limit of two. It would additionally make it easier to dynamically create components, by manually retyping a set of untyped kernel objects and tweaking the capability access-control in accordance with the minimum requirements for different executable and linkable format (ELF) sections — e.g., read-only for `.rodata`, read-only and executable for `.text`. Instead, we chose to develop the TEE with CAmkES for its ease of use and automated, reliable capability distribution.

We identify multiple directions for future work. First would be the implementation of dynamic loading of TAs, building upon our design in Section 4.2.1 and the implementation considerations in Section 5.4. This could further show that our remote attestation security guarantees not only apply to a locked down system, but additionally extend to TEEs that are more usable. This is important, because ease of use is crucial for adoption of security.

A second direction is to extend the performance evaluation of our TEE with benchmarks and measurements. Useful metrics could include the time to attest a certain area of memory, and benchmark results from the PARSEC suite [12]. This would further prove our reasoning of the small overhead for remote attestation.

8. Related work

"I kind of lost track of time..."
"For two hours?"
"There were books involved."

Brandon Sanderson, *The Well of Ascension*

The increasing use of trusted execution environments (TEEs) has also lead to an increase in respective security research. The publication of the Spectre and Meltdown speculative side-channel attacks instigated many new research directions that seek to defend against these software-based attacks, and to provide isolated execution and secure remote attestation. More generally, there is much work on defending against microarchitectural side-channel attacks.

Mushtaq et al. [60] survey countermeasure techniques for cache-based side-channel attacks. The countermeasures range from new hardware designs, to software and application-specific mitigations. They include techniques for partitioning caches [91, 24, 21], time slicing caches [97, 90, 36, 96], and disabling resource sharing [78, 46, 20, 98]. Though these techniques provide mitigations for the attacks caused by microarchitectural flaws, they still rely on shared hardware. In contrast, our thesis evades the vulnerability to these types of flaws by separating sensitive operations from the main processor altogether.

SANCTUARY [13] extends Arm TrustZone with user space enclaves to enable execution of security-critical applications with strong isolation. However, the system operates on the same processor in secure world, which makes SANCTUARY vulnerable to side-channel attacks on the L2 cache. Varys [65] is a system to protect applications running in Intel Software Guard Exentsions (SGX) enclaves from side-channel attacks, by limiting the sharing of resources such as L1 and L2 caches. However, CacheOut [89] bypasses the Varys software defenses.

Recent work [31, 94] explores the feasibility of resetting the microarchitectural state — a.o. flushing cache upon security boundary switches — to mitigate side-channel attacks. The results are inconclusive but promising for wider application.

HYDRA [27] is a remote attestation design built with the seL4 microkernel. It relies on secure boot functionality to verify that a correct seL4 microkernel image is booted. HYDRA uses the capability-based access control of seL4 to achieve its security properties for remote attestation. This is all implemented in software, unlike prior designs

[23, 51] that relied on hardware implementations. The software executables (analogous to trusted applications (TAs) in our work) run as separate processes in seL4, but it remains unclear how they are deployed to the system. HYDRA's adversarial model explicitly states that side-channel attacks are out of scope. Additionally, its attestation procedures and storage of cryptographic secrets all occur inside of seL4. Reasoning from current knowledge, this makes it vulnerable to transient execution attacks. Our design addresses these concerns by offloading the sensitive data and operations to a secure co-processor.

VRASED [63] is another design for remote attestation, that uses both hardware and software. VRASED focuses on formal verification of the remote attestation scheme. It achieves this by modeling the sub-modules of the hardware unit as finite state machines (FSMs) in languages that allow to verify them against models, and by using a formally-verified cryptography library for the software unit [99]. APEX [64] extends VRASED by providing provable execution of attested software. The main difference is that VRASED and APEX do not provide an isolated execution environment for attested software. By using seL4, our work continues to provide integrity and isolation of software after it has been attested. Additionally, VRASED and APEX target resource-constrained devices, whereas our work also extends to more powerful mobile devices.

The Sancus security architecture [62] for resource-constrained devices provides remote attestation and isolation guarantees, among other properties. Developers can deploy software to the platform, similar to our TEE. The Sancus architecture extends the processor core with hardware units for cryptographic operations and memory access control. In contrast to Sancus, our work uses a separate co-processor and has formal verification of the isolation mechanisms by relying on the seL4 microkernel.

Apple's Secure Enclave [5] and Google's Titan M [86] are examples of existing secure co-processors deployed on a large scale. They are isolated from the main processor and designed to be secure even when the main processor becomes compromised. Apple's Secure Enclave has a boot ROM to provide a hardware root of trust (RoT), a dedicated engine for efficient cryptographic operations, a random number generator, and protected memory. Google's Titan M chips comprise similar components, but additionally have embedded flash memory. The functionalities of these co-processors are consistent with those formulated in our system model (Section 3.1). An implementation of our TEE can thus leverage these pre-existing co-processors.

seL4 currently only targets static architectures. However, the seL4 foundation stated that seL4 will eventually support late loading of components [40]. The dynamic loading of TAs in the TEE can then be implemented using this functionality.

9. Conclusion

"What is fate but coincidences in retrospect?"

Ken Liu, *The Grace of Kings*

In this work we designed and developed a trusted execution environment (TEE) that provides remote attestation integrity protection even when confidentiality of the TEE is compromised. Transient execution side-channel attacks compromise confidentiality of attestation keys in existing TEEs, thereby breaking trust of the remote attestation procedure. We protected against this by performing the sensitive cryptographic operations on a secure co-processor, which does not share any vulnerable microarchitectural hardware units with the main processor. We built the TEE using the formally verified seL4 microkernel that provides strong isolation and integrity. We can extend our work to leverage co-processors from Google and Apple, for wide-scale deployment on mobile devices. Our design and prototype show that providing secure remote attestation for TEEs is possible by using a separate co-processor to guarantee integrity and a formally verified microkernel for strongly isolated execution.

References

[1] Tigist Abera et al. “C-FLAT: Control-Flow Attestation for Embedded Systems Software”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS’16: 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna Austria: ACM, Oct. 24, 2016, pp. 743–754. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978358. URL: https://dl.acm.org/doi/10.1145/2976749.2978358 (visited on June 21, 2021).

[2] Dakshi Agrawal et al. “The EM Side—Channel(s)”. In: *Cryptographic Hardware and Embedded Systems - CHES 2002*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2523. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 29–45. ISBN: 978-3-540-00409-7. DOI: 10.1007/3-540-36400-5_4. URL: http://link.springer.com/10.1007/3-540-36400-5_4 (visited on May 26, 2021).

[3] Ittai Anati et al. *Innovative Technology for CPU Based Attestation and Sealing*. Intel Corporation, Aug. 14, 2013. URL: https://software.intel.com/content/www/us/en/develop/articles/innovative-technology-for-cpu-based-attestation-and-sealing.html.

[4] James P. Anderson. *Computer Security Technology Planning Study. Volume 2:* Fort Belvoir, VA: Defense Technical Information Center, Oct. 1, 1972. DOI: 10.21236/AD0772806. URL: http://www.dtic.mil/docs/citations/AD0772806 (visited on May 24, 2021).

[5] *Apple Platform Security*. Apple, May 2021. 219 pp. URL: https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf.

[6] *ARM Security Technology: Building a Secure System Using TrustZone Technology*. Arm Ltd., 2009.

[7] *Armv8-M Architecture Reference Manual*. DDI0553B.o. Arm Ltd., 2021. URL: https://developer.arm.com/documentation/ddi0553/bo.

[8] *Armv8, for Armv8-A Architecture Profile*. DDI0487G.a. Arm Ltd., Jan. 2021.

[9] *Basic Architecture*. Vol. 1. 10 vols. Intel® 64 and IA-32 Architectures Software Developer’s Manual. Intel Corporation, Apr. 2021.

[10] Daniel J Bernstein et al. “High-Speed High-Security Signatures”. In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89. DOI: 10.1007/s13389-012-0027-1.

[11] Daniel J. Bernstein et al. “TweetNaCl: A Crypto Library in 100 Tweets”. In: *International Conference on Cryptology and Information Security in Latin America*. Ed. by Diego F. Aranha and Alfred Menezes. Vol. 8895. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 64–83. ISBN: 978-3-319-16294-2. DOI: 10.1007/978-3-319-16295-9_4. URL: http://link.springer.com/10.1007/978-3-319-16295-9_4 (visited on June 24, 2021).

[12] Christian Bienia et al. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT ’08. Toronto, Ontario, Canada: ACM Press, 2008, pp. 72–81. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128. URL: http://portal.acm.org/citation.cfm?doid=1454115.1454128 (visited on July 5, 2021).

[13] Ferdinand Brasser et al. “SANCTUARY: ARMing TrustZone with User-Space Enclaves”. In: *26th Annual Network and Distributed System Security Symposium*. NDSS ’19. San Diego, CA: Internet Society, 2019. ISBN: 978-1-891562-55-6. DOI: 10.14722/ndss.2019.23448. URL: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_01A-1_Brasser_paper.pdf (visited on June 30, 2021).

[14] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *Computer Networks* 48.5 (Aug. 2005), pp. 701–716. ISSN: 13891286. DOI: 10.1016/j.comnet.2005.01.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128605000125 (visited on Jan. 18, 2021).

[15] Jo Van Bulck et al. “FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Security 18. 2018, pp. 991–1008.

[16] Claudio Canella et al. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security 19. Santa Clara, CA: USENIX Association, 2019, pp. 249–266.

[17] David Cerdeira et al. “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-Assisted TEE Systems”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. May 2020, pp. 1416–1432. DOI: 10.1109/SP40000.2020.00061.

[18] Ricardo Chaves et al. “Improving SHA-2 Hardware Implementations”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Ed. by Louis Goubin and Mitsuru Matsui. Red. by David Hutchison et al. Vol. 4249. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 298–310. ISBN: 978-3-540-46559-1. DOI: 10.1007/11894063_24. URL: http://link.springer.com/10.1007/11894063_24 (visited on June 24, 2021).

[19] Guoxing Chen et al. “SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution”. In: *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. June 2019, pp. 142–157. DOI: 10.1109/EuroSP.2019.00020.

[20] David Cock et al. “The Last Mile: An Empirical Study of Timing Channels on seL4”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS’14: 2014 ACM SIGSAC Conference on Computer and Communications Security. Scottsdale Arizona USA: ACM, Nov. 3, 2014, pp. 570–581. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660294. URL: https://dl.acm.org/doi/10.1145/2660267.2660294 (visited on June 30, 2021).

[21] Patrick Colp et al. “Protecting Data on Smartphones and Tablets from Memory Attacks”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’15: Architectural Support for Programming Languages and Operating Systems. Istanbul Turkey: ACM, Mar. 14, 2015, pp. 177–189. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694380. URL: https://dl.acm.org/doi/10.1145/2694344.2694380 (visited on June 30, 2021).

[22] Karl De Leeuw and Jan Bergstra. *The History of Information Security: A Comprehensive Handbook*. Elsevier, 2017.

[23] Karim El Defrawy et al. “SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust”. In: *19th Annual Network and Distributed System Security Symposium*. NDSS ’12. 2012, pp. 1–15.

[24] Leonid Domnitser et al. “Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks”. In: *ACM Transactions on Architecture and Code Optimization* 8.4 (Jan. 2012), pp. 1–21. ISSN: 1544-3566, 1544-3973. DOI: 10.1145/2086696.2086714. URL: https://dl.acm.org/doi/10.1145/2086696.2086714 (visited on June 30, 2021).

[25] Zakir Durumeric et al. “The Matter of Heartbleed”. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC ’14: Internet Measurement Conference. Vancouver BC Canada: ACM, Nov. 5, 2014, pp. 475–

488. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663755. URL: https://dl.acm.org/doi/10.1145/2663716.2663755 (visited on June 7, 2021).

[26] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. "The Untapped Potential of Trusted Execution Environments on Mobile Devices". In: *IEEE Security & Privacy* 12.4 (July 2014), pp. 29–37. ISSN: 1558-4046. DOI: 10.1109/MSP.2014.38.

[27] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. "HYDRA: Hybrid Design for Remote Attestation (Using a Formally Verified Microkernel)". In: *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec '17: 10th ACM Conference on Security & Privacy in Wireless and Mobile Networks. Boston Massachusetts: ACM, July 18, 2017, pp. 99–110. ISBN: 978-1-4503-5084-6. DOI: 10.1145/3098243.3098261. URL: https://dl.acm.org/doi/10.1145/3098243.3098261 (visited on June 1, 2021).

[28] *Exynos 1080 5G Mobile Processor: Specs, Features | Samsung Exynos*. Samsung Semiconductor. URL: //www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-1080/ (visited on June 8, 2021).

[29] Erhu Feng et al. "Scalable Memory Protection in the PENGLAI Enclave". In: *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*. USENIX OSDI 21. USENIX Association, July 2021, p. 20.

[30] Andrei Frumusanu. *Qualcomm Details The Snapdragon 888: 3rd Gen 5G & Cortex-X1 on 5nm*. URL: https://www.anandtech.com/show/16271/qualcomm-snapdragon-888-deep-dive (visited on June 8, 2021).

[31] Qian Ge, Yuval Yarom, and Gernot Heiser. "No Security Without Time Protection: We Need a New Hardware-Software Contract". In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. APSys '18: 9th Asia-Pacific Workshop on Systems. Jeju Island Republic of Korea: ACM, Aug. 27, 2018, pp. 1–9. ISBN: 978-1-4503-6006-7. DOI: 10.1145/3265723.3265724. URL: https://dl.acm.org/doi/10.1145/3265723.3265724 (visited on June 24, 2021).

[32] Qian Ge et al. "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware". In: *Journal of Cryptographic Engineering* 8.1 (Apr. 2018), pp. 1–27. ISSN: 2190-8508, 2190-8516. DOI: 10.1007/s13389-016-0141-6. URL: http://link.springer.com/10.1007/s13389-016-0141-6 (visited on May 26, 2021).

[33] Qian Ge et al. "Time Protection: The Missing OS Abstraction". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19: Fourteenth EuroSys Conference 2019. Dresden Germany: ACM, Mar. 25, 2019, pp. 1–17. ISBN:

978-1-4503-6281-8. DOI: 10.1145/3302424.3303976. URL: https://dl.acm.org/doi/10.1145/3302424.3303976 (visited on June 5, 2021).

[34] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis". In: *Annual Cryptology Conference*. Springer, 2014, pp. 444–461.

[35] Daniel Gruss et al. "KASLR Is Dead: Long Live KASLR". In: *Engineering Secure Software and Systems*. Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos. Vol. 10379. Cham: Springer International Publishing, 2017, pp. 161–176. ISBN: 978-3-319-62104-3. DOI: 10.1007/978-3-319-62105-0_11. URL: http://link.springer.com/10.1007/978-3-319-62105-0_11 (visited on Feb. 22, 2021).

[36] Daniel Gruss et al. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna Austria: ACM, Oct. 24, 2016, pp. 368–379. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978356. URL: https://dl.acm.org/doi/10.1145/2976749.2978356 (visited on June 30, 2021).

[37] Shay Gueron. *A Memory Encryption Engine Suitable for General Purpose Processors*. 2016/204. Cryptology ePrint Archive, 2016.

[38] Lachlan Gunn. *Graphical Analysis of TEE-Based Systems*. Unpublished. Aalto University, 2020.

[39] David Harris and Sarah L. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2010.

[40] Gernot Heiser. "The seL4 Report: An Update from seL4 Land". FOSDEM '21. Feb. 2021. URL: https://fosdem.org/2021/schedule/event/microkernel_sel4_report/attachments/slides/4482/export/events/attachments/microkernel_sel4_report/slides/4482/heiser.pdf (visited on June 28, 2021).

[41] Gernot Heiser, Toby Murray, and Gerwin Klein. "Towards Provable Timing-Channel Prevention". In: *ACM SIGOPS Operating Systems Review* 54.1 (Aug. 31, 2020), pp. 1–7. ISSN: 0163-5980. DOI: 10.1145/3421473.3421475. URL: https://dl.acm.org/doi/10.1145/3421473.3421475 (visited on June 5, 2021).

[42] *Intel's 11th Gen Processor (Rocket Lake-S) Architecture Detailed*. Intel. Oct. 2020. URL: https://www.intel.com/content/www/us/en/newsroom/news/11th-gen-processor-rocket-lake-s-architecture.html (visited on June 8, 2021).

[43] *Intel's Pentium 4 Prescott Processor*. The Tech Report. Feb. 2, 2004. URL: https://techreport.com/review/6213/intels-pentium-4-prescott-processor/ (visited on May 28, 2021).

[44] Trent Jaeger. "Operating System Security". In: *Synthesis Lectures on Information Security, Privacy, and Trust* 1.1 (Jan. 2008), pp. 1–218. ISSN: 1945-9742, 1945-9750. DOI: 10.2200/S00126ED1V01Y200808SPT001. URL: http://www.morganclaypool.com/doi/abs/10.2200/S00126ED1V01Y200808SPT001 (visited on May 24, 2021).

[45] Simon Johnson et al. *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*. Intel Corporation, Mar. 9, 2016, p. 10. URL: https://software.intel.com/content/www/us/en/develop/download/intel-sgx-intel-epid-provisioning-and-attestation-services.html.

[46] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud". In: *Proceedings of the 21st USENIX Security Symposium*. USENIX Security 12. USENIX Association, 2012, pp. 189–204.

[47] Gerwin Klein et al. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Transactions on Computer Systems* 32.1 (Feb. 2014), pp. 1–70. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/2560537. URL: https://dl.acm.org/doi/10.1145/2560537 (visited on June 5, 2021).

[48] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09*. The ACM SIGOPS 22nd Symposium. Big Sky, Montana, USA: ACM Press, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: http://portal.acm.org/citation.cfm?doid=1629575.1629596 (visited on June 8, 2021).

[49] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Annual International Cryptology Conference*. Springer, 1999, pp. 388–397. DOI: 10.5555/646764.703989.

[50] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[51] Patrick Koeberl et al. "TrustLite: A Security Architecture for Tiny Embedded Devices". In: *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*. The Ninth European Conference. Amsterdam, The Netherlands: ACM Press, 2014, pp. 1–14. ISBN: 978-1-4503-2704-6. DOI: 10.

1145/2592798.2592824. URL: http://dl.acm.org/citation.cfm?doid=2592798.2592824 (visited on July 1, 2021).

[52] Ihor Kuz et al. “CAmkES: A Component Model for Secure Microkernel-Based Embedded Systems”. In: *Journal of Systems and Software*. Component-Based Software Engineering of Trustworthy Embedded Systems 80.5 (May 1, 2007), pp. 687–699. ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.08.039. URL: https://www.sciencedirect.com/science/article/pii/S016412120600224X (visited on June 7, 2021).

[53] Ihor Kuz et al. “capDL: A Language for Describing Capability-Based Systems”. In: *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems - APSys ’10*. The First ACM Asia-Pacific Workshop. New Delhi, India: ACM Press, 2010, p. 31. ISBN: 978-1-4503-0195-4. DOI: 10.1145/1851276.1851284. URL: http://portal.acm.org/citation.cfm?doid=1851276.1851284 (visited on Feb. 8, 2021).

[54] Dayeol Lee et al. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20: Fifteenth EuroSys Conference 2020. Heraklion Greece: ACM, Apr. 15, 2020, pp. 1–16. ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3387532. URL: https://dl.acm.org/doi/10.1145/3342195.3387532 (visited on Feb. 8, 2021).

[55] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *Communications of the ACM* 63.6 (May 21, 2020), pp. 46–56. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3357033. URL: https://dl.acm.org/doi/10.1145/3357033 (visited on Feb. 22, 2021).

[56] Brian McGillion et al. “Open-TEE – An Open Virtual Trusted Execution Environment”. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. Aug. 2015, pp. 400–407. DOI: 10.1109/Trustcom.2015.400.

[57] Frank McKeen et al. “Innovative Instructions and Software Model for Isolated Execution”. In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP ’13*. Vol. 10. Tel-Aviv, Israel: ACM Press, 2013. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488368. URL: http://dl.acm.org/citation.cfm?doid=2487726.2488368 (visited on July 10, 2021).

[58] *MultiZone Security for RISC-V*. Hex Five Security. June 30, 2019. URL: https://hex-five.com/multizone-security-sdk/ (visited on June 3, 2021).

[59] Kit Murdock et al. “Plundervolt: Software-Based Fault Injection Attacks against Intel SGX”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. May 2020, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057.

[60] Maria Mushtaq et al. “Winter Is Here! A Decade of Cache-Based Side-Channel Attacks, Detection & Mitigation for RSA”. In: *Information Systems* 92 (Sept. 1, 2020), p. 101524. ISSN: 0306-4379. DOI: 10.1016/j.is.2020.101524. URL: https://www.sciencedirect.com/science/article/pii/S0306437920300338 (visited on June 1, 2021).

[61] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. *A Survey of Published Attacks on Intel SGX*. June 24, 2020. arXiv: 2006.13598 [cs]. URL: http://arxiv.org/abs/2006.13598 (visited on May 31, 2021).

[62] Job Noorman et al. “Sancus 2.0: A Low-Cost Security Architecture for IoT Devices”. In: *ACM Transactions on Privacy and Security* 20.3 (Aug. 11, 2017), pp. 1–33. ISSN: 2471-2566, 2471-2574. DOI: 10.1145/3079763. URL: https://dl.acm.org/doi/10.1145/3079763 (visited on July 1, 2021).

[63] Ivan De Oliveira Nunes, Karim Eldefrawy, and Norrathep Rattanavipanon. “VRASED: A Verified Hardware/Software Co-Design for Remote Attestation”. In: *Proceedings of the 28th USENIX Security Symposium*. USENIX Security 19. USENIX Association, 2019, pp. 1429–1446.

[64] Ivan De Oliveira Nunes et al. “APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise”. In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Security 20. USENIX Association, 2020, pp. 771–788.

[65] Oleksii Oleksenko et al. “Varys: Protecting SGX Enclaves From Practical Side-Channel Attacks”. In: *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX ATC 18. Boston, MA: USENIX Association, July 2018, pp. 227–240. ISBN: 978-1-939133-01-4.

[66] H. Orman. “The Morris Worm: A Fifteen-Year Perspective”. In: *IEEE Security & Privacy* 1.5 (Sept. 2003), pp. 35–43. ISSN: 1558-4046. DOI: 10.1109/MSECP.2003.1236233.

[67] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *Topics in Cryptology – CT-RSA 2006*. Ed. by David Pointcheval. Red. by David Hutchison et al. Vol. 3860. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20. ISBN: 978-3-540-31033-4. DOI: 10.1007/11605805_1. URL: http://link.springer.com/10.1007/11605805_1 (visited on May 31, 2021).

[68] Mingliang Pei et al. *Trusted Execution Environment Provisioning (TEEP) Architecture*. Internet-Draft draft-ietf-teep-architecture-14. Internet Engineering Task Force, Feb. 22, 2021, p. 34. URL: https://datatracker.ietf.org/doc/html/draft-ietf-teep-architecture-14.

[69] Sandro Pinto and Nuno Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Computing Surveys* 51.6 (Feb. 27, 2019), pp. 1–36. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3291047. URL: https://dl.acm.org/doi/10.1145/3291047 (visited on June 2, 2021).

[70] *Samsung Knox*. White paper. Samsung. URL: https://docs.samsungknox.com/admin/whitepaper/kpe/samsung-knox.htm (visited on June 3, 2021).

[71] *seL4 Reference Manual*. 12.0.0. Trustworthy Systems Team, Data61, Nov. 6, 2020, p. 186. URL: https://sel4.systems/Info/Docs/seL4-manual-12.0.0.pdf (visited on June 5, 2021).

[72] *seL4/Camkes-Manifest*. seL4 Foundation, June 11, 2021. URL: https://github.com/seL4/camkes-manifest (visited on July 4, 2021).

[73] *seL4/Camkes-Tool*. seL4 Foundation, June 2, 2021. URL: https://github.com/seL4/camkes-tool (visited on June 7, 2021).

[74] *seL4/Capdl*. seL4 Foundation, Apr. 27, 2021. URL: https://github.com/seL4/capdl (visited on June 7, 2021).

[75] *seL4/Global-Components*. seL4 Foundation, Mar. 8, 2021. URL: https://github.com/seL4/global-components (visited on July 4, 2021).

[76] *seL4/L4v*. seL4 Foundation, June 7, 2021. URL: https://github.com/seL4/l4v (visited on June 9, 2021).

[77] *seL4/seL4-CAmkES-L4v-Dockerfiles*. seL4 Foundation, July 2, 2021. URL: https://github.com/seL4/seL4-CAmkES-L4v-dockerfiles (visited on July 4, 2021).

[78] Jicheng Shi et al. "Limiting Cache-Based Side-Channel in Multi-Tenant Cloud Using Dynamic Page Coloring". In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. June 2011, pp. 194–199. DOI: 10.1109/DSNW.2011.5958812.

[79] François-Xavier Standaert. "Introduction to Side-Channel Attacks". In: *Secure Integrated Circuits and Systems*. Ed. by Ingrid M.R. Verbauwhede. Integrated Circuits and Systems. Boston, MA: Springer US, 2010, pp. 27–42. ISBN: 978-0-387-71827-9. DOI: 10.1007/978-0-387-71829-3_2. URL: http://link.springer.com/10.1007/978-0-387-71829-3_2 (visited on May 26, 2021).

[80] László Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[81] Andrew S. Tanenbaum. *Modern Operating Systems*. Fourth edition. Boston: Pearson, 2015. 1101 pp. ISBN: 978-0-13-359162-0.

[82] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *Proceedings of the 26th USENIX Security Symposium*. USENIX Security 17. Aug. 2017, pp. 1057–1074.

[83] *TEE Protection Profile*. GlobalPlatform, July 2020. URL: https://globalplatform.org/specs-library/tee-protection-profile-v1-3/.

[84] *TEE System Architecture v1.2*. GlobalPlatform, Nov. 2018. URL: https://globalplatform.org/specs-library/tee-system-architecture-v1-2/.

[85] David Thomson. *Understanding Samsung Knox Vault: Protecting the Data That Matters Most*. Samsung Business Insights. Mar. 8, 2021. URL: https://insights.samsung.com/2021/03/08/understanding-samsung-knox-vault-protecting-data/ (visited on June 3, 2021).

[86] *Titan M Makes Pixel 3 Our Most Secure Phone Yet*. Google. Oct. 17, 2018. URL: https://blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/ (visited on June 9, 2021).

[87] Furkan Turan and Ingrid Verbauwhede. "Compact and Flexible FPGA Implementation of Ed25519 and X25519". In: *ACM Transactions on Embedded Computing Systems* 18.3 (June 13, 2019), pp. 1–21. ISSN: 1539-9087, 1558-3465. DOI: 10.1145/3312742. URL: https://dl.acm.org/doi/10.1145/3312742 (visited on June 24, 2021).

[88] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *2020 IEEE Symposium on Security and Privacy (SP)*. May 2020, pp. 54–72. DOI: 10.1109/SP40000.2020.00089.

[89] Stephan van Schaik et al. *CacheOut: Leaking Data on Intel CPUs via Cache Evictions*. June 23, 2020. arXiv: 2006.13353 [cs]. URL: http://arxiv.org/abs/2006.13353 (visited on May 28, 2021).

[90] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. "Scheduler-Based Defenses against Cross-VM Side-Channels". In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security 14. San Diego, CA: USENIX Association, 2014, pp. 687–702.

[91] Zhenghong Wang and Ruby B Lee. "New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, 2007, pp. 494–505.

[92] Andrew Waterman et al., eds. *User-Level ISA, Document Version 20191213*. Vol. 1. The RISC-V Instruction Set Manual. RISC-V International.

[93] *Widevine DRM - Getting Started with Devices*. Google, 2018. URL: https://web.archive.org/web/20190504082905/https://storage.googleapis.com/wvdocs/Widevine_DRM_Getting_Started_Devices.pdf (visited on June 4, 2021).

[94] Nils Wistoff et al. *Prevention of Microarchitectural Covert Channels on an Open-Source 64-Bit RISC-V Core*. May 1, 2020. arXiv: 2005.02193 [cs]. URL: http://arxiv.org/abs/2005.02193 (visited on June 24, 2021).

[95] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *Proceedings of the 23rd USENIX Security Symposium*. USENIX Security 14. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. ISBN: 978-1-931971-15-7.

[96] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 346–367. ISBN: 978-3-662-53140-2.

[97] Yinqian Zhang and Michael K. Reiter. "Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*. The 2013 ACM SIGSAC Conference. Berlin, Germany: ACM Press, 2013, pp. 827–838. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516741. URL: http://dl.acm.org/citation.cfm?doid=2508859.2516741 (visited on June 30, 2021).

[98] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. "A Software Approach to Defeating Side Channels in Last-Level Caches". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna Austria: ACM, Oct. 24, 2016, pp. 871–882. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978324. URL: https://dl.acm.org/doi/10.1145/2976749.2978324 (visited on June 30, 2021).

[99] Jean-Karim Zinzindohoué et al. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas Texas USA: ACM, Oct. 30, 2017, pp. 1789–1806. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134043. URL:

https://dl.acm.org/doi/10.1145/3133956.3134043 (visited on June 29, 2021).