SoK: Hardware-supported Trusted Execution Environments

Moritz Schneider
ETH Zurich

Ramya Jayaram Masti
Intel Cooperation

Shweta Shinde
ETH Zurich

Srdjan Capkun
ETH Zurich

Ronald Perez
Intel Cooperation

***Abstract*—The growing complexity of modern computing platforms and the need for strong isolation protections among their software components has led to the increased adoption of Trusted Execution Environments (TEEs). While several commercial and academic TEE architectures have emerged in recent times, they remain hard to compare and contrast. More generally, existing TEEs have not been subject to a holistic systematization to understand the available design alternatives for various aspects of TEE design and their corresponding pros-and-cons.**

Therefore, in this work, we analyze the design of existing TEEs and systematize the mechanisms that TEEs implement to achieve their security goals, namely, verifiable launch, run-time isolation, trusted IO and secure storage. More specifically, we analyze the typical architectural building blocks underlying TEE solutions, design alternatives for each of these components and the trade-offs that they entail. We focus on hardware-assisted TEEs and cover a wide range of TEE proposals from academia and the industry. Our analysis shows that although TEEs are diverse in terms of their goals, usage models and instruction set architectures, they all share many common building blocks in terms of their design.

I. Introduction

Today's computing platforms are diverse in terms of their architectures, software provisioning models, and the types of applications they support. They include large-scale servers used in cloud computing, smartphones used in mobile networks, and smart home devices. Over time, these devices have evolved to store and process security-sensitive data to enable novel applications in various domains such as the healthcare and financial industry. Therefore, modern computing platforms must implement mechanisms to protect such security-sensitive data against unauthorized access and modification.

Data confidentiality and integrity solutions in today's computing systems have to not only account for attacks over the network, but also those that originate from (a subset of) software and hardware components on the same platform or from an adversary with physical access. This is because these systems typically host multiple, mutually-untrusted software components. Examples of such software deployment models include code/data from different tenants that share the same cloud platform and code/data belonging to users and network providers that share the same smartphone. Therefore, platforms today have to isolate sensitive data from potentially co-resident software and hardware adversaries. In fact, this requirement equally extends to any computations that involve such sensitive data and, more generally, any security-critical computations.

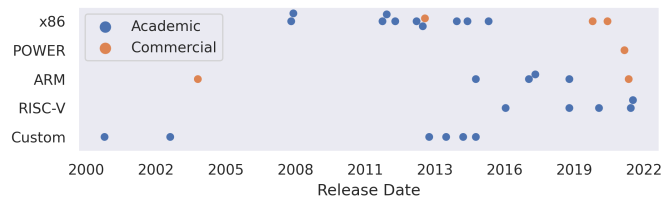


Fig. 1: The release dates of the surveyed TEEs according to their instruction set architecture.

An increasingly popular solution to the problem of protecting sensitive computations and data against co-located attackers is a *Trusted Execution Environment (TEE)*. While existing TEEs often vary in terms of their exact security goals, most of them aim to provide (a subset of) four high level security protections, namely, *(i)* verifiable launch of the execution environment for the sensitive code and data so that a remote entity can ensure that it was set up correctly, *(ii)* run-time isolation to protect the confidentiality and integrity of sensitive code and data, *(iii)* trusted IO to enable secure access to peripherals and accelerators, and finally, *(iv)* secure storage for TEE data that must be stored persistently and made available only to authorized entities at a later point in time.

Today, a variety of academic and commercial TEE designs and solutions exist. They are diverse in terms of their underlying security assumptions, their target instruction set architectures and the usage models they support (Fig. 1). In this work, we analyze the design of existing TEEs and systematize common as well as unique design decisions of TEEs. Even though many TEE designs use different names and descriptions of their underlying mechanisms, the resulting mechanisms are often very similar. To reason about the similarities between these designs, we group the underlying mechanisms into classes of mechanisms and highlight exceptional cases.

We begin our study of TEE solutions by identifying and classifying the common set of adversaries that most TEEs consider, based on which platform components they control in software and hardware. We focus on hardware-assisted TEEs and cover how they implement verifiable launch, run-time CPU and memory isolation, trusted IO and secure storage. Analyzing which of these features different TEEs support, how they implement them, and which attackers they consider allows us to compare a wide range of TEEs spanning different usages including mobile and cloud computing. It also enables us to be inclusive and cover a vast majority of existing academic and commercial TEEs.

The first TEE security goal we study is verifiable launch which refers to providing proof regarding the correctness of the initial state of the TEE. This is typically achieved by first establishing a Root of Trust for Measurement (RTM), then leveraging it to *measure* the state of the code/data within the TEE, and finally, making this available for verification through a process called *attestation*. Standard measurement and attestation processes have long been established by the Trusted Computing Group using a Trusted Platform Module (TPM) [1]. To a large extent, today's attestation solutions involve similar mechanisms and protocols but have evolved over time to rely on different architectural components. For example, in some TEEs, attestation keys and measurements are held in on-chip components [2]–[4], but others rely on the off-chip TPM [5], [6]. Key hierarchies involved in modern attestation protocols are also different from standard TPM-based protocols and, in some cases, involve symmetric keys for within-platform attestation for performance reasons.

We then focus on how TEEs implement run-time isolation to protect the confidentiality and integrity of sensitive computations and data. We introduce a taxonomy of isolation strategies that classifies them according to two dimensions: resource partitioning and isolation enforcement. We then use these dimensions to reason about the techniques used by individual TEE designs. As each strategy has unique (dis-)advantages for every resource, we discuss their suitability for isolating CPU and memory against different adversaries. Then, we summarize if and how different TEE solutions adopt these strategies by describing the various architectural components they use for CPU and memory isolation. We show that despite their diversity, most modern TEE designs use a single common strategy for CPU protection. In contrast, the strategies for memory protection are diverse, with some TEEs employing different strategies against different adversaries.

Trusted IO solutions for use with TEEs have evolved over time from supporting user IO to more diverse accelerators. Most trusted IO solutions involve two main components: a trusted path to the device and a trusted device architecture to protect security sensitive data just like the CPU. We identify two common types of trusted paths that TEEs can implement, namely logical and cryptographic. Then, we describe different ways to implement these trusted paths, their suitability for use in different scenarios and the architectural support they require in each case. Finally, we apply the same taxonomy of isolation strategies that we used in the context of CPU and memory isolation to understand diverse trusted device architectures.

Secure storage in the context of TEEs involves ensuring that any sensitive data that is persistent is only available to authorized entities. This concept is often referred to as *sealing*. Similar to measurement and attestation solutions, early sealing mechanisms for TEEs typically rely on concepts pioneered by the TPM [1]. We observe that sealing processes have adapted over time to cater to new requirements (e.g., migration, anti-rollback) as well as rely only on on-chip elements. Our study reveals that only about a third of existing TEE solutions discuss sealing support with similar implementation approaches.

Overall, this paper makes the following contributions towards an improved understanding of TEE architectures.

- This paper describes an adversary model including software and physical attackers and their capabilities. We believe that this taxonomy of adversaries is useful both during the design of TEEs in terms of choosing between different security mechanisms as well as for analyzing their security. This stems from the fact that the choice of security mechanisms in TEE solutions often differ not only by the resource being protected but also by the type of adversary being thwarted.
- To the best of our knowledge, this is the first effort to identify and classify the design decisions made by TEEs to achieve four fundamental security goals, namely verifiable launch, run-time isolation, trusted IO, and secure storage. We believe that this systematization can be used as a basis for designing new TEE architectures based on different design constraints and security models.
- Based on the surveyed TEE architectures, we conclude that the design space of the four fundamental security goals to be relatively small. New proposals usually re-use many design choices and only propose minor modifications to a single component.

II. Scope

Numerous new TEE architectures have been proposed in recent years. Oddly, despite the abundant research on TEEs and the growing number of commercially available TEE solutions, there is no single, widely-accepted definition for a TEE. In this paper, we do not attempt to find such a general definition of a TEE. Instead, we survey a wide variety of existing approaches and systematize them according to their architectural support of four security properties common in all of them: (i) verifiable launch, (ii) run-time isolation, (iii) trusted IO, and (iv) secure storage. With all the differences in existing TEE designs, we aim to find underlying design decisions that connect all these seemingly different approaches. Since the performance of a TEE is mainly determined by the processor design and not the TEE specifics, we do not investigate performance differences between the surveyed TEEs.

The selection of designs to survey is critical to this paper and it is based on the following criteria:

- We focus on hardware-assisted TEEs and do not investigate arguably more complex software approaches, e.g., relying on trusted hypervisors.
- We consider TEEs from many different processor architectures. We selected designs covering four major processor architectures: x86, ARM, POWER, and RISC-V. At the same time we also analyzed proposals based on more niche instruction set architectures such as SPARC and OpenRISC.
- While some proposals are sometimes not regarded to be TEEs (e.g., ARM TrustZone) we still attempt to include

as many designs that may fit in the envelope of a TEE even if the proposal itself does not use the term "TEE".

- We deliberately exclude proposals based on co-processors such as Google Titan and Apple's Secure Enclaves, or proposals based on hardware security modules (HSM). We want to focus on approaches that run on general-purpose processors and are closely intertwined with other untrusted applications running on the same processor.

Terminology: In academic literature and the industry, numerous names have been used for the various components of a TEE. For example, Intel Software Guard Extensions (SGX) refers to TEE instances as *enclaves* [7], Trust Domain Extensions (TDX) uses *Trust Domains* [8], AMD Secure Encrypted VMs-Secure Nested Paging (SEV-SNP) [3] uses *Secure Encrypted VMs*, ARM Confidential Compute Architecture (CCA) uses *Realms* for their VM-isolation solution and *trustlets* for their application isolation feature [9]. In this paper, we use *enclave* to refer to such a TEE instance and trusted computing base (*TCB*) to describe all underlying trusted components. We use *TEE* to refer to the entire architecture that enables the creation of enclaves.

III. System and Adversary Models

This section discusses the generic platform underlying most TEE designs and the types of adversaries they consider.

A. System Model

Most TEE solutions target a modern general-purpose computing platform that consists of a System-on-Chip (SoC) with off-chip memory and, optionally, off-chip peripherals, as shown in Fig. 2. The SoC itself contains one or more cores that potentially share a cache (or a part thereof) and fabric that connects them to the memory controller. SoCs also include an IO complex that is used to connect both on-chip and off-chip peripherals to the SoC fabric and caches. The typical software stack of the system includes an operating system (OS) and multiple userspace applications. If virtualized, a hypervisor runs underneath one or more Virtual Machines (VMs), each with their own (guest) OS and userspace applications. These software components run at different privilege levels on most modern CPUs (see Fig. 3). The hardware and software components that are used to achieve the security protections of a TEE is called the Trusted Computing Base (TCB).

B. Adversary Model

TEEs aim to provide a variety of security protections against a wide range of adversaries that are co-resident on the same platform. These include untrusted co-resident software (e.g., code in other enclaves, system management software such as an OS), untrusted platform hardware (e.g., IO peripherals), a hardware attacker who has physical access to the platform (e.g., bus interposers) or a combination thereof. We discuss these adversaries and the types of attacks they can launch with respect to a *victim enclave* (see Fig. 2).

Co-located enclave adversary: This adversary is relevant when the platform supports more than one enclave, either concurrently or over time. This adversary is capable of launching one or more enclaves with code/data of its choice. Such a situation commonly emerges, for example, in multi-tenant cloud platforms where multiple users can launch enclaves on shared hardware. It also occurs in mobile ecosystems where multiple service providers provision code (and data) to run within enclaves. For brevity, we use A_{tee} to refer to such an attacker as well as any code/resources that it controls.

Unprivileged software adversary: This adversary can launch any *unprivileged* software on the same shared hardware as the victim enclave. Such an attacker can often run code with the same privilege level as the victim's enclave but not higher. Examples of such adversaries include cloud users that control guest VMs in cloud environments that run alongside the victim enclave, and mobile phone users that can launch apps to run concurrently with mobile phone enclaves. We use A_{app} to refer to such an attacker.

System software adversary: This refers to an adversary that controls the system management software, such as an OS that manages the platform's resources. This adversary has all the capabilities of A_{tee} and A_{app}. Additionally, it controls system resources such as memory and scheduling. Hence, it is more powerful than the above adversaries. Examples of such adversaries include the untrusted hypervisor in cloud settings and the OS running on a mobile phone. This adversary is referred to as A_{ssw}.

Startup adversary: This refers to an adversary that controls the system boot of the platform that hosts the TEE. Such an attacker controls system configuration such as memory and IO fabric parameters that could undermine the entire TEE design if misconfigured. Examples of such an attacker include an untrusted BIOS. This adversary is represented as A_{boot}.

Peripheral adversary: Modern platforms could include multiple peripherals that are not in the TCB of the victim enclave. These peripherals could be within the SoC or the connected over to it over an external bus. They are often assumed to be untrusted, especially if they include firmware that could be potentially exploited remotely. An adversary controlling such a peripheral can launch nefarious IO transactions to try to access or modify memory and other resources belonging to the victim enclave. We use A_{per} to denote such an attacker.

Fabric adversary: Another adversary considered by TEE architectures has the ability to introduce special hardware such as fabric interposers to launch man-in-the-middle attacks [10]. The fabric adversary can also directly access data-at-rest such as the disk or external memory. However, this adversary cannot breach the SoC package: everything within the package remains out-of-scope. In the rest of the paper, we use A_{bus} to represent this adversary.

Invasive adversary: This adversary can launch invasive attacks such as de-layering the physical chip, manipulating clock signals and voltage rails to cause faults, etc., to extract secrets or force a different execution path than the intended one. For the sake of completeness, we include this adversary

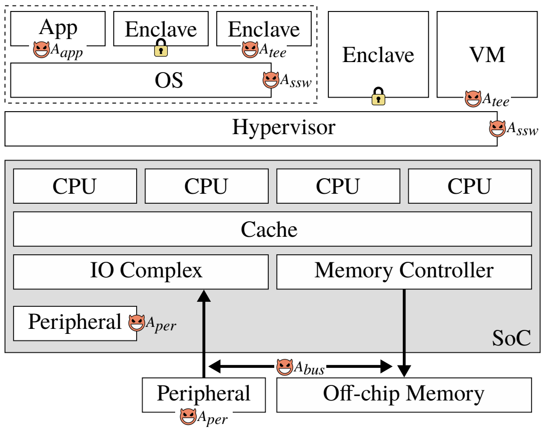


Fig. 2: The software and hardware system model with our adversary model. The boot adversary (A_{boot}) and the invasive adversary (A_{inv}) are omitted.

	x86	ARM	RISC-V	PowerPC	PL
App / App / App / App	Ring 3	EL0	U	PR	PL3
Guest OS / Guest OS	Ring 0	EL1	S	OS	PL2
Hypervisor	Ring -1	EL2	H	Hyp	PL1
Firmware	Ring -2	EL3	M	-	PL0

Fig. 3: Summary of privilege levels in modern processors: Most CPUs support at least four privilege levels, one each for userspace applications, an OS, a hypervisor. In this paper, we will use PL0-PL3 shown in grey to denote them.

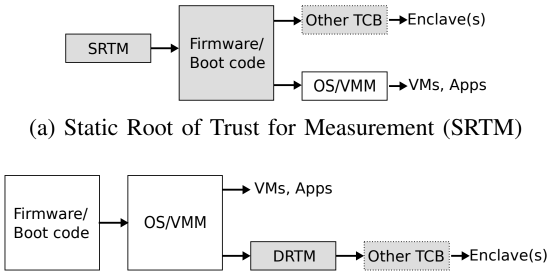
(A_{inv}) in our list but note that no TEE design currently defends against such an attacker. So, we do not discuss this attacker any further in this paper.

C. A Note on Side-Channel Attacks

Numerous physical [11], [12] and micro-architectural [13]–[15] side-channel attacks have been explored and shown to be feasible on modern systems both in the context of TEEs [14] as well more broadly [13]. Such attacks can be launched by any of the above adversaries with varying success. Given that side-channels are often a result of shared computation resources and not often specific to enclaves, TEEs often rely on generic countermeasures to mitigate their impact. For example, generic software defense approaches such as eliminating secret-dependent memory accesses [16] or secret dependent branching [17] equally apply for use with enclaves. Today, most (commercial) TEE proposals explicitly exclude side-channel attacks from their attacker model and recommend using existing countermeasures to protect against them. Even though side-channel attacks are not the focus of this paper, we will still briefly mention the impact of specific design decisions on side-channels where it is appropriate.

IV. Verifiable Launch

A critical first step that precedes the actual execution of an enclave is a secure setup process that ensures that the enclave's execution environment is configured correctly and that its initial state is as expected. The prevalent verifiable boot process in TEEs is enclave measurement and attestation. Intuitively, a measurement of an enclave, and more generally any software, is a fingerprint of its initial state, typically constructed by a series of one or more cryptographic hashes over the initial memory of the enclave. The measurement process itself must be trustworthy; it begins at the Root-of-Trust for Measurement (RTM) and is implemented as a chain-of-trust through the enclave's TCB, which finally measures the enclave itself. The measurements are used later as part of a digitally signed report sent to a verifier through a process referred to as attestation. Attestation could provide additional information about the enclave's security properties (e.g., the authenticity of the platform hosting the enclave, details about its TCB itself). This section discusses architectural support for different types of RTMs, typical measurement processes, and attestation schemes of TEEs.



(a) Static Root of Trust for Measurement (SRTM)

(b) Dynamic Root of Trust for Measurement (DRTM)

Fig. 4: Types of Root of Trust for Measurement (RTM) used by modern TEEs: The RTM may directly measure the enclave or optionally measure and launch one or more TCB components that eventually measure the enclave. The chain-of-trust for measurement (shown in grey) consists of the RTM and any intermediate TCB components that are eventually responsible for the enclave's final measurement.

A. Root of Trust (RTM)

Central to a verifiable launch process is an RTM, which serves as the trust anchor for the measurement process. Currently, TEEs use one of three types of RTMs, namely, static (SRTM), dynamic (DRTM), and hardware based (HW), as summarized in Table I.

SRTM is created by an unbroken chain of trust from the code module that first executes at system reset to the code that runs in the enclave. This chain of trust usually only includes all the software components of the enclave's TCB, as shown in Fig. 4. The chain typically does not include the operating system. Such a solution can bootstrap the run-time components of the system TCB before any untrusted components (A_{tee}, A_{app}, A_{ssw}, and A_{per}) are even active. For solutions that consider A_{bus}, such bootstrapping must ensure that no off-chip components (e.g., platform TPM) are required during this process. A typical SRTM could be implemented entirely in hardware or immutable software in a BootROM.

In contrast, solutions that use a DRTM can establish a new RTM without trusting the code executed prior to it since system reset (see Fig. 4b). So, these solutions must implement architectural extensions to protect the bootstrap process against adversaries (A_{tee}, A_{app}, A_{ssw}, and A_{per}) that may potentially be active at the time of enclave launch [5], [6], [18]. This can be done through specialized hardware instructions that first suspend all other active processes (hence, A_{tee}, A_{app}, A_{ssw}) and disable all IO devices and interrupts (hence, A_{per}). Then, the hardware loads, measures, and authenticates a signed-code module that serves as the TCB of the actual enclave. Once the hardware has verified the signed-code module and potentially recorded its measurement, it executes the module that in turn loads and measures the enclave itself.

Most surveyed TEEs use SRTM (Table I). Only a few systems from two commercial processor manufacturers (Intel and AMD) leverage DRTM. We speculate that the main motivation of DRTM — excluding boot code from the TCB — only applies to platforms with a large amount of legacy boot code (e.g., x86 BIOS [19]).

B. Measurement

In SRTM and DRTM solutions, each entity in the chain of trust up to the enclave, starting at the RTM, measures the next component before transferring execution control to it. In practice, all components in such a chain of trust are not only measured but also integrity-checked (e.g., by verifying a signature, checking the measurement against a reference value) before they are executed. We note that all surveyed TEEs use very similar techniques for measurement and we did not discover major differences. We refer the reader to Appendix A1 for a further discussion on implementation details of a typical measurement mechanism.

C. Architectural Support for Attestation

Attestation is the third and final step of verifiable launch, where a verifier checks that the enclave has been launched correctly and that its initial state is as expected. More specifically, the verifier ensures that the enclave's measurement and its underlying TCB match their expected reference values. There are two flavors of attestation: local attestation and remote attestation. Local attestation is applicable when a verifier is co-located with the enclave on the same platform. In contrast, remote attestation is meant for use by a remote verifier that is not on the same platform as the enclave being attested. Remote attestation schemes usually rely on asymmetric cryptography and often incur the cost of checking one or more certificate chains. This can be expensive, especially when implemented in hardware. In contrast, local attestation is typically implemented using symmetric cryptography and tends to be more efficient. Most existing TEE solutions support remote attestation (Table I), but only a handful specify both local and remote attestation, as summarized in Table I. Note that the remote attestation of some TEEs can trivially be reused for local attestation.

	Name	ISA	RTM	Attestation	
				Local	Remote
Industry	Intel SGX [7], [20]	x86	DRTM	●	●
	Intel TDX [8]	x86	DRTM	●	●
	AMD SEV-SNP [3]	x86	SRTM	○	●
	ARM TZ [21]	ARM	SRTM	○	○
	ARM Realms [9]	ARM	SRTM	○	●
	IBM PEF [22]	POWER	SRTM	●	●
Academia	Flicker [5]	x86	DRTM	○	●
	SEA [6]	x86	DRTM	○	●
	SICE [23]	x86	SRTM	○	●
	PodArch [24]	x86	SRTM	○	○
	HyperCoffer [25]	x86	SRTM	○	○
	H-SVM [26], [27]	x86	SRTM	○	○
	EqualVisor [18]	x86	SRTM	○	○
	xu-cc15 [28]	x86	SRTM	○	○
	wen-cf13 [29]	x86	SRTM	○	○
	Komodo [30]	ARM	SRTM	○	●
	SANCTUARY [31]	ARM	SRTM	●	●
	TrustICE [32]	ARM	SRTM	○	○
	HA-VMSI [33]	ARM	SRTM	○	●
	Sanctum [4]	RISC-V	SRTM	●	●
	TIMBER-V [34]	RISC-V	SRTM	●	●
	Keystone [35]	RISC-V	SRTM	○	●
	Penglai [36]	RISC-V	SRTM	○	●
	CURE [37]	RISC-V	SRTM	○	●
	Iso-X [38]	OpenRISC	SRTM	○	●
	HyperWall [39]	SPARC	SRTM	○	●
	Sancus [40], [41]	MSP430	HW	○	●
	TrustLite [42]	Custom	SRTM	●	●
	TyTan [43]	Custom	SRTM	●	●
	XOM [44]	Custom	SRTM	○	○
	AEGIS [45]	Custom	SRTM	○	○

TABLE I: The surveyed TEEs with their respective Root-of-Trust for Measurement (RTM) and their support for local and remote attestation. We use SRTM for static Root-of-Trust, DRTM for dynamic Root-of-Trust, and HW for hardware based systems that do not rely on SRTM or DRTM (c.f., Section IV-A). ● indicates a TEE that describes a specific attestation mechanism whereas ○ is used for TEEs with no mention of such a mechanism. We note that some remote attestation schemes can be trivially re-used for local attestation.

D. Provisioning Secrets into an Enclave

Provisioning secrets into enclaves is often the last optional step during its launch. Some TEEs such as IBM PEF [22], AMD SEV-SNP [3], PodArch [24], and Wen-cf13 [29] allow enclaves to be provisioned with secret data prior to the attestation. In this case, the enclave's initial state will contain some secret values also reflected in the measurement. This is achieved by an enclave provisioning mechanism where the developer encrypts the secret data before delivering the enclave binary to the platform.

Other TEE designs require attestation before any secret data can be provisioned. To establish a communication channel bound to an attestation report, enclaves in these TEEs may append some custom data (e.g., a public key certificate) to the attestation report [2], [3], [46]. Since this extra data is also authenticated during the attestation, its integrity is protected and can be leveraged to construct entire secure channels

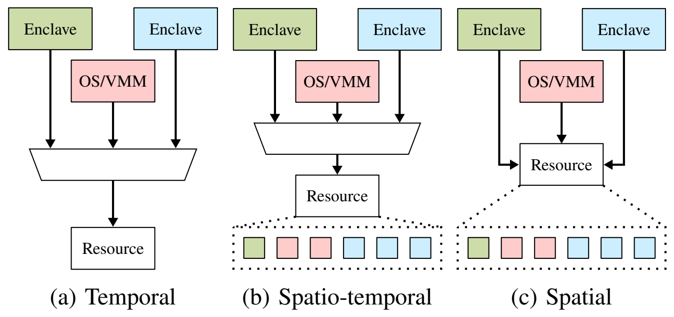


Fig. 5: Resources can be partitioned temporally, spatially, or a mix thereof (spatio-temporal).

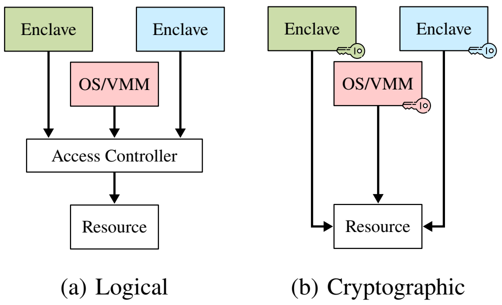


Fig. 6: Isolation enforcement strategies.

based on a key exchange protocol of choice. We note that AMD SEV [3] supports both the initial secret provisioning and establishing a secure channel bound to an attestation.

V. Run-time Isolation

Following the setup of the enclave, it begins executing. In order to prevent attackers from interfering with enclave execution, all the resources belonging to the enclave, including its CPU and memory, must be protected against unauthorized access and tampering. Such protection mechanisms are typically referred to as *run-time isolation*. Below, we first describe a broad taxonomy of isolation strategies. Then, we discuss if and how one could apply them for CPU and memory isolation. Finally, we survey the set of strategies used by existing TEE designs to protect CPU and memory.

A. Taxonomy of Isolation Strategies

In general, isolation mechanisms aim to achieve confidentiality and integrity of the protected resource, and can broadly be classified according to how resources are partitioned and how isolation is enforced. We describe these two dimensions below and depict them in Figs. 5 and 6.

Partitioning Resources: Resources can be isolated in space, time, or a mix thereof (Fig. 5). Note that this is not a categorical classification but rather a smooth range from fully temporal to fully spatial. As we will discuss later, these extreme cases of fully temporal or spatial only rarely appear. Instead, most isolation mechanism have some temporal and spatial aspects, i.e., they are spatio-temporal.

Temporal partitioning splits a resource in the time domain, i.e., it securely multiplexes the same resource among multiple execution contexts over time. At any point in time, a single execution context has exclusive access to the resource. Temporal partitioning requires mechanisms to securely switch contexts while re-assigning the resource to a new execution context. Such secure context switches should be fast to not impact the general system performance. Temporal isolation is often used for resources that are costly to spatially partition and situations where concurrent access from multiple execution contexts to the same resource is not required.

In *spatial* partitioning, resources are split such that trusted and untrusted contexts use separate, dedicated partitions. It can be used when there are multiple identical copies of the same resource (e.g., logical processor) or when the resource can be split into smaller identical copies. Note that a given execution context may be assigned more than one instance of a resource (e.g., multiple logical processors) if required. As a result, spatial isolation techniques are often used for resources that are relatively cheap to replicate or to split. It also entails implementing mechanisms to ensure that the different copies of the resource can be used concurrently by different entities without any interference among them.

Spatio-temporal partitioning leverages both temporal as well as spatial aspects to partition a resource, e.g., the resource can be spatially partitioned but these partitions may change over time. This concept can only be used in resources that support both temporal and spatial partitioning. However, it may provide more flexibility and some performance advantages, and thus, is quite a popular choice.

Enforcement: In contrast to resource partitioning, enforcement of isolation strategies can be classified into two distinct categories: logical and cryptographic isolation. An overview of the two strategies is depicted in Fig. 6.

Logical isolation leverages logical access control mechanisms to enforce isolation. These mechanism prohibit the adversary from accessing protected data. For example, a trusted context switch routine uses logical isolation to make sure the next execution thread cannot access any protected data by saving and purging the processor registers. On the other hand, many logical isolation mechanisms intercept data accesses and check the requests against some access control information. This access control information must be generated and managed by a trusted entity in the system, and it could be modified at run-time to enable flexible resource re-allocation. The resources (e.g., storage) needed to maintain the access control information varies depending on the granularity of the access control information and the type of resource being managed. Furthermore, the access control information itself must be protected against attacks.

As the name indicates, *cryptographic isolation* uses cryptography to achieve isolation. Confidentiality is usually achieved via encryption; only authorized contexts with access to the correct cryptographic key material can decrypt data correctly. In contrast to logical isolation where protected data is not accessible at all, unauthorized contexts may read the ciphertext but they cannot retrieve the plaintext. Integrity is in part achieved through a cryptographic Message Authentication Code (MAC) stored alongside the data. This prevents an unauthorized context from tampering with data that does not belong to it because it cannot generate the correct MAC for a given piece of data without access to the correct keys. Achieving complete integrity with cryptographic isolation requires using anti-replay schemes that prevent re-injection of old data (with the correct, corresponding MAC) at a later point in time.

Below, we discuss the application of the above isolation strategies to CPU and memory and the resulting trade-offs.

B. CPU Isolation

While a typical CPU has many components, the discussion below focuses on the architectural state within the CPU such as the register state. This is mainly because many details on micro-architectural CPU state (e.g., intermediate CPU buffers, schedulers) are not publicly available for commercial TEE solutions. Even academic TEE solutions often omit these details. However, the impact of TEE implementations on memory-related micro-architectural structures (e.g., caches, TLBs) in the CPU are available and are analyzed in Section V-C.

Choice of CPU Isolation Strategy: While all isolation strategies are applicable to the architectural state within the CPU, most of them have considerable downsides, e.g., spatially reserving a (virtual) core exclusively for an enclave incurs sub-optimal resource utilization and limits the number of concurrent enclaves. Similarly, spatio-temporal approaches require extra hardware in a performance-critical part of the processor to protect spatially separated data. Hence, these techniques are not well suited for CPU isolation. In contrast, temporal partitioning is very well suited for CPU isolation as it does not add additional runtime checks beside a trusted context switch.

On the enforcement side, cryptographic enforcement suffers from a large performance overhead due to the extra cryptographic hardware on the fast-path of the processor. On the other hand, temporal partitioning combined with logical enforcement does not exhibit such overheads besides requiring a fast and secure context switching routine that temporally separates multiple execution contexts on the same CPU thread.

In-line with our analysis, all existing TEE solutions that we studied use temporal partitioning combined with logical enforcement for CPU state. Our results are listed in Table II.

Architectural Support for CPU Isolation: As mentioned before, temporal partitioning with logical enforcement is typically implemented through a secure context switch routine that saves, purges, and restores the execution contexts. The context switch routine must ensure that the data from an enclave does not leak to any untrusted context or another enclave that follows it in the execution schedule. In addition, an untrusted context must not be able to tamper with or, more generally, control the CPU execution state of an enclave when it is being started or resumed. While there are multiple options to save and restore execution contexts such as encrypting the registers on enclave exit, most TEEs save the context to the enclave's private memory and afterwards purge the register values. To achieve this, TEEs rely on their TCB for setting up the CPU state correctly before starting an enclave and scrubbing the CPU state while exiting an enclave.

In order to ensure that the TCB can fully mediate every context switch, TEEs leverage multiple CPU modes, privilege levels, and in some cases, a combination thereof. This ensures that all transitions into and out of an enclave occur from TCB-controlled mode(s)/privilege level(s). The actual solution used by a TEE design often depends on the underlying processor's instruction set architecture. Commercial processors from Intel, AMD, ARM, and IBM often add new execution modes to support TEEs (see Fig. 7). In contrast, most academic TEEs do not introduce any new processor modes or privilege levels. Instead, they rely on the firmware running in an existing high-privilege level (PL0) for secure context switching. There are a few proposals in which the hardware itself facilitates the temporal isolation during the context switch [27]–[29], [38], [39], [44], [45].

Besides context switching, CPU modes and privilege levels are often necessary for securely running the enclave and its software TCB components (if any), as summarized in Table II. While the enclave code typically encompasses an application or a virtual machine and runs at lower privilege (PL2, PL3), the TCB tends to run at higher privilege in most TEE designs (PL0, PL1). This allows the TCB to implement many other types of security mechanisms such as measurement and attestation as discussed in Section IV as well as isolation of memory, trusted IO, and secure storage as discussed in the rest of this paper.

C. Memory Isolation

Ensuring that the memory used by an enclave is protected at run-time against unauthorized access and modification is a particularly challenging aspect of TEE design. Such protections must cover not only the actual off-chip memory, but also any code/data that resides in the on-chip micro-architectural structures such as instruction and data caches. Furthermore, since most TEEs support virtual memory, trustworthiness (or lack thereof) of the translation structures such as page tables that convert virtual addresses to physical addresses are a key design aspect of all memory isolation solutions. Similar to processor caches, translation look-aside buffers (TLB) holding recent page translations must also be protected against misuse and misconfiguration.

Choice of Memory Isolation Strategy: All previously discussed isolation strategies can be applied to memory and we discuss the implications of these strategies below. While all TEE designs used temporal partitioning for CPU isolation, their memory isolation strategies are diverse. In fact, many TEE designs use different strategies based on the type(s) of attacker(s) under consideration, as shown in Fig. 8.

Full spatial partitioning of memory implies reserving one or more memory regions for exclusive use by an enclave or the TCB and these regions remain assigned to it until the next system reboot. Spatial partitioning works well when the number of enclaves that the system must support is small,

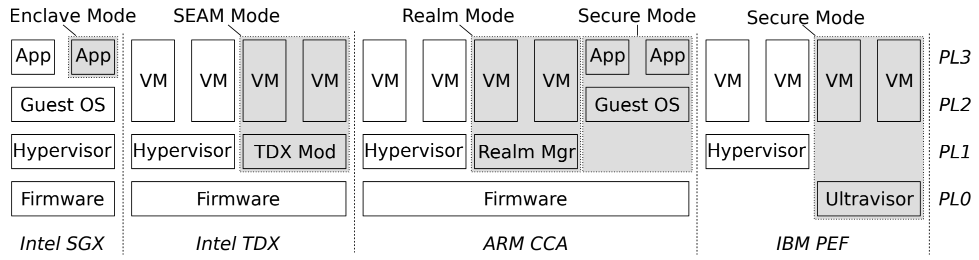


Fig. 7: Some TEEs introduce new CPU modes with their design, e.g., Intel SGX, Intel TDX, ARM CCA, and IBM PEF.

	Name	Isol Strat	Privilege Level	
			Enclave	Software TCB
Industry	Intel SGX [7], [20]	T-L	App	-
	Intel TDX [8]	T-L	VM	PL1
	AMD SEV-SNP [3]	T-L	VM	-[†]
	ARM TZ [21]	T-L	App/VM	PL0+(PL1/2)[‡]
	ARM CCA [9]	T-L	VM	PL0+(PL1)[‡]
	IBM PEF [22]	T-L	VM	PL0
Academia	Flicker [5]	T-L	VM	-
	SEA [6]	T-L	VM	-
	SICE [23]	T-L	VM	PL0
	PodArch [24]	T-L	App	-
	HyperCoffer [25]	T-L	VM	PL0
	H-SVM [26], [27]	T-L	VM	-
	EqualVisor [18]	T-L	VM	PL1
	xu-cc15 [28]	?	App	-
	wen-cf13 [29]	T-L	VM	-
	Komodo [30]	T-L	App	PL0 + PL2
	SANCTUARY [31]	T-L	App	PL0 + PL2
	TrustICE [32]	T-L	App	PL0 + PL2
	HA-VMSI [33]	T-L	VM	PL0
	Sanctum [4]	T-L	App	PL0
	TIMBER-V [34]	T-L	App	PL0
	Keystone [35]	T-L	App	PL0
	Penglai [36]	T-L	App	PL0
	CURE [37]	T-L	App/VM	PL0
	Iso-X [38]	T-L	App	-
	HyperWall [39]	T-L	VM	-
	Sancus [40], [41]	T-L	App	-
	TrustLite [42]	T-L	App	PL0
	TyTan [43]	T-L	App	PL0
	XOM [44]	T-L	App	-
	AEGIS [45]	T-L	App	-

[†] AMD SEV-SNP palces the TCB in separate co-processor [3].
[‡] ARM TZ and ARM Realms only provide the hardware primitives to implement a TEE. There are multiple options to implement the software TCB in different privilege levels.

TABLE II: Summary of CPU Isolation in TEEs: All TEE solutions use temporal and logical isolation (indicated by T-L) to securely share the CPU among execution contexts. Enclaves are run in either an App (PL3) or as a VM (PL2). The software TCB is implemented in a more privileged level than the enclave.

and their memory resource requirements are fairly static and well-known in advance. It also works well for coarse-grained memory protections that protect access control information used for logical isolation (as explained below).

Fully temporal memory partitioning involves allowing memory accesses only from the currently active execution context and securely saving/restoring memory content during context switches. Its use for memory isolation is limited to scenarios where only a single execution context is active at any point in time. Hence, it is not efficient for TEEs supporting concurrent enclaves. Besides, saving memory content to disk can take considerable time. Thus, temporal memory partitioning is rarely used.

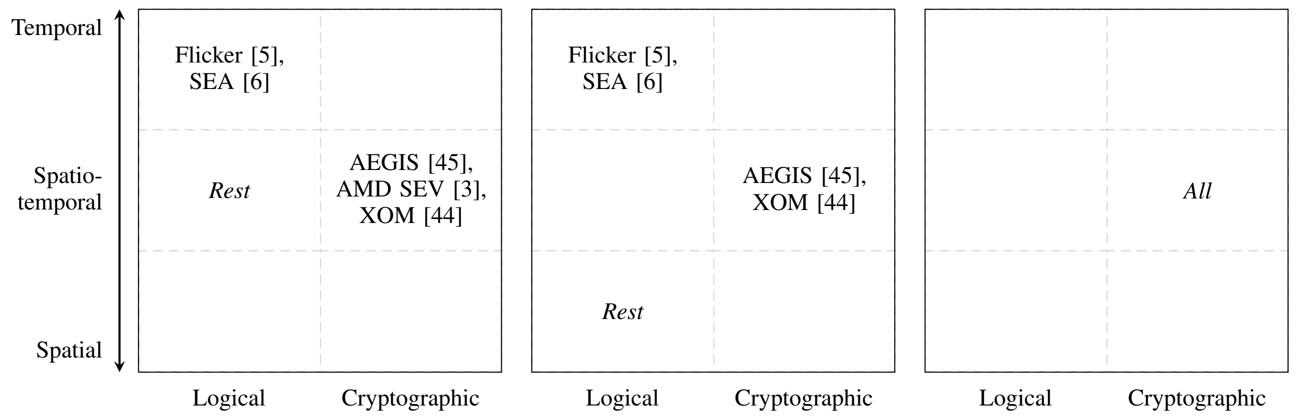
Spatio-temporal memory partitioning is the preferred style due to its flexibility: memory regions can be re-allocated to a different execution context over time. Hence, it is useful in systems where memory requirements for enclaves cannot be predicted upfront.

Logical enforcement relies on access control mechanism to only allow authorized accesses to enclave resources. Logical isolation requires every access to be checked against access control information, e.g., by a memory management unit (MMU). Furthermore, achieving integrity protection against software adversaries is rather efficient, with only a small amount of access control information per enclave.

Cryptographic enforcement achieves confidentiality through encryption. It ensures integrity by maintaining a cryptographic MAC for each block of memory of configurable size. Protection against replay attacks is achieved by maintaining freshness information (e.g., counters), often in the form of a Merkle tree [45], [47]. In contrast to all the isolation strategies above, cryptographic memory isolation can protect against A_{bus}. However, cryptographic isolation is hard to scale, especially if different execution contexts need separate keys, because it requires maintaining large amounts of cryptographic keying material on-die within the SoC. Furthermore, achieving integrity and anti-replay properties using cryptographic isolation results in storage overheads (e.g., for the MAC and anti-replay metadata) as well as latency and throughput overheads [47].

In the surveyed TEEs, we found a variety of such strategies. While there often is a preferred design choice (indicated by *Rest* or *all* in Fig. 8), other options exist and seem to be practical. We highlight that many TEEs use multiple strategies simultaneously, e.g., Intel SGX [7] uses spatio-temporal partitioning and logical enforcement to protect enclave memory, but it uses purely spatial partitioning to protect its TCB, and cryptographic isolation to cope with A_{bus}.

Architectural Support for Memory Isolation: Logical enforcement of spatial or spatio-temporal memory isolation is facilitated by an access control check. All surveyed TEEs use one of two options for the access control check: memory protection units (MPU) or memory management units (MMU).



(a) Enclave memory isolation against A_{app}, A_{tee}, and A_{ssw}.

(b) TCB memory isolation against A_{app}, A_{tee}, and A_{ssw}.

(c) Memory isolation strategy against A_{bus}.

Fig. 8: The isolation strategies employed in main memory according to the adversaries they protect against. Note that many TEEs use distinct strategies for different adversaries. Not all surveyed TEEs support a physical adversary (c.f., Table IV in the appendix).

We refer the reader to Appendix B for a discussion on the implementation details of these two options. One of the main differences between MPU and MMU based enforcement, is that the former operates on physical addresses and the latter on virtual addresses. Also, MPUs typically only support a limited number of rules, whereas MMUs are more flexible. We note that TEEs that leverage MMUs are typically more complex and often come with an in-depth security analysis. On the other hand, MPUs are rather simple and thus may simplify the security analysis. Many modern academic TEEs rely on an MPU to provide isolation [4], [31], [34], [35], [37]. On the other hand, many commercial TEEs seem to appreciate the increased flexibility of the MMU [3], [7], [9], [22].

We note that the access control information used to enforce memory isolation, i.e., trusted page tables for an MMU, access control rules for the MPU, or other secondary metadata, must itself be protected against unauthorized access. This is done typically through spatial partitioning, i.e., memory for such structures is allocated at boot and protected through a simplified MPU (e.g., range registers).

Caches: Caches contain recently used portions of memory for improved software performance. Often, CPUs contain multiple cache layers, some exclusive to a core and others that are shared. In some TEE architectures, the MPU/MMU in the CPU and their IO counterparts prevent untrusted entities from accessing enclave data in the caches (e.g., Intel SGX). In other TEEs where such accesses cannot be prevented by existing mechanisms, additional cache protection mechanisms isolate enclave data in the caches (e.g., Arm TrustZone). A summary of the isolation strategies for caches is depicted in Fig. 9.

Spatial cache partitioning, i.e., reserving portions of the cache for exclusive use by an enclave, does not scale well, reduces resource utilization, and can lead to potential performance degradation. However, it can be used to mitigate side-channel attacks [4], [35]–[37], [48]. Temporal cache partitioning is not very efficient because it requires flushing the entire cache on every transition among execution contexts. Hence, it is only used by a handful of TEE designs to protect against side-channel leakage and is limited to small caches that are exclusive to a single core [4], [31], [35]. Cryptographic enforcement is not suitable for micro-architectural structures like caches because of the cost due to extra cryptographic hardware as well as its limited latency and throughput. So, most caches in TEE solutions today implement spatio-temporal and logical cache isolation.

VI. Trusted IO

While early TEE designs only focus on the CPU, recent interest in secure interactions with external devices has inspired multiple approaches for trusted IO [37], [49]–[51]. Trusted IO has two components: *(i)* confidentiality and integrity for the enclave's accesses to the device, i.e., establishing a *trusted path*, and *(ii)* protecting enclave data on the device through a *trusted device architecture*. We discuss existing solutions for each of these components below.

A. Establishing a Trusted Path

Originally, the term *trusted path* was used in the context of enabling trusted IO interactions for users [52]. However, today, with the increasing diversity of IO devices and the trend towards enabling their use with TEEs, a trusted path is also used to refer to the (secure) communication channel between an enclave and a device.

Both logical and cryptographic isolation techniques can be used to establish a trusted path. If the trusted path has multiple hops, each hop could implement a different type of isolation. While both logical and cryptographic trusted paths can protect against A_{tee}, A_{app}, A_{ssw}, A_{boot} and A_{per}, only cryptographic trusted paths can protect against A_{bus}. Below, we discuss different ways to implement logical and cryptographic trusted paths.

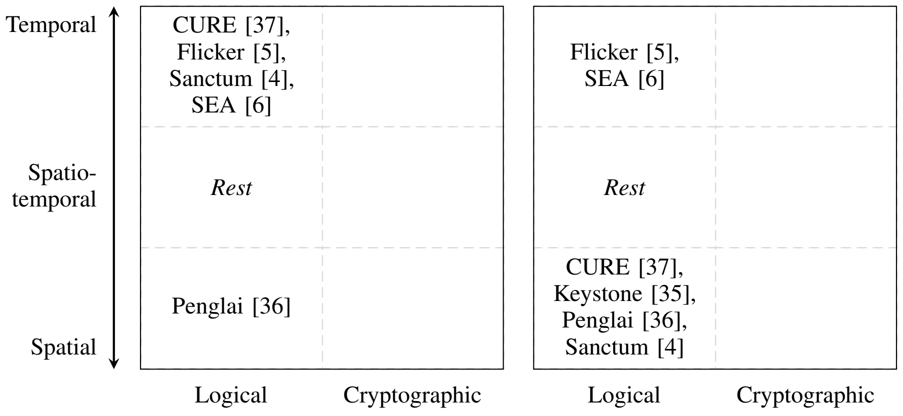


Fig. 9: The isolation strategies employed in local and shared caches against software adversaries (A_{app}, A_{tee}, and A_{ssw}). Most TEEs use spatio-temporal logical isolation. However, some TEEs leverage fully temporal or spatial partitioning.

Architectural Support for Logical Trusted Path: Logical trusted paths require architectural support to enable two types of IO interactions: direct-memory-access (DMA) and memory mapped IO (MMIO). In this discussion, we focus on MMIO and refer the reader to Section V-C for a detailed analysis of isolation mechanisms for DMA.

One way to build a logical trusted path for MMIO is through access control filters that allow/deny accesses based on the origin or the destination of the MMIO request. Such filters could be static or programmable at run-time by a trusted entity. Many systems rely on ARM's TrustZone Protection Controller to filter accesses to peripherals [49], [50], [53], [54]. CURE [37], TrustOTP [55] and HectorV [51] rely on similar but more complex filters in front of each peripheral to allow/disallow accesses. Another option to build a logical trusted path is through trusted memory mappings via secure MPU configurations (e.g., TrustLite [42]) or related metadata structures (e.g., HIX [56]) that are checked every time an enclave tries to access a device.

Architectural Support for Cryptographic Trusted Path: End-to-end cryptographic trusted paths rely on a secure channel established between two endpoints: the enclave and the device. This approach incurs overhead related to the cryptographic operations and hardware. To establish a secure channel, the device and enclave must be provisioned with credentials and cryptographic keys to use for authentication and optionally, attestation. Examples of solutions that use a cryptographic trusted path include Fidelius [57] and HIX [56]. Sometimes, cryptographic channels may be multi-hop, i.e., include a trusted intermediate hardware component between the enclave and the device (e.g., Bastion SGX [58], ProtectIOn [59], and HETEE [60]). Cryptographic trusted paths at the link level (as opposed to high software levels) are emerging to protect specifically against A_{bus} [61].

Certain trusted path solutions combine both cryptographic and logical trusted paths. For example, SGXIO [62] uses a cryptographic path between the CPU package and the device, but isolates accesses from different enclaves using a trusted intermediary on the CPU package with exclusive access to the device. It is also possible to use a different type of trusted path for the DMA and MMIO respectively; HIX [56] uses a logical trusted path for MMIO and a cryptographic trusted path for DMA.

B. Trusted Device Architectures

For certain types of trusted IO usages, it is sufficient to establish a trusted path from the enclave to the device. Examples include peripherals that do not process user data in clear text (e.g., encrypted storage devices, network cards). However, newly emerging devices such as accelerators are used for computations on user data. Such accelerators must ensure the confidentiality and integrity of every enclave's data just as the CPU does.

Today, a variety of accelerators exist: e.g., custom chips for AI processing, Field Programmable Logic Arrays (FPGAs) and general purpose Graphics Processing Units (GPUs). These systems differ greatly in terms of their underlying architectures and hence, the exact mechanisms that they must implement to isolate enclave data also varies. The details of these mechanisms for individual accelerators are out-of-scope for this paper. However, there are still a set of high-level isolation techniques based on the strategies discussed in Section V-A that apply to many of these accelerators as discussed below.

Spatial Partitioning: Assigning separate/dedicated instances of an accelerator to each enclave for the entire lifetime of the platform is typically not resource or cost efficient. Therefore, while it is feasible to do in theory, spatial isolation is not very practical. However, if such dedicated instances are available, then establishing a trusted path to the device suffices and no additional device requirements arise.

Temporal Partitioning: Until recently, accelerators were built assuming exclusive use by a single execution context at any given time. Therefore, temporal partitioning, i.e., sharing it among multiple contexts over time, is a common approach. Such temporal partitioning requires a secure context switching mechanism which could be implemented either in software or by enhancing the accelerator hardware itself. Example solutions that rely on such temporal partitioning include HETEE [60] for generic single-use accelerators, ZeroKernel [63] and HIX [56] for GPUs, as well as MeetGo [64] and ShEF [65] for FPGAs.

Spatio-temporal Partitioning and Logical Enforcement: This is typically used when multiple enclaves need concurrent access to an accelerator due to its flexibility. Such spatio-temporal isolation requires hardware support by the device architecture to enable true multi-tenancy. Again, the exact set of enhancements are device-specific, but they often involve maintaining access control information to track ownership of resources, i.e., mapping of resources to enclaves. Examples of such solutions include Graviton [66], Telekine [67] and SEGIVE [68] for GPUs as well as Trustore [69] for FPGAs.

Cryptographic Enforcement: This is not well-suited for protecting on-chip accelerator resources (e.g., caches, TLBs) for the same reasons as it is not optimal for protecting on-chip CPU resources, namely, due to the performance overhead as well as additional area cost for all the cryptography hardware. However, cryptographic enforcement can be applied specifically to device-side memory resources just like they apply to DRAM on the CPU as described in Section V.

VII. Secure Storage

In many applications, enclaves are required to retain certain (persistent) state across different invocations. The process of protecting the data in this way through encryption is referred to as *sealing* and the reverse (decryption) process that accounts for enclave state is called *unsealing*.

While such secure storage is a very common requirement, it is not described explicitly by most existing TEE designs. Some TEEs support primitives that can be leveraged for secure storage (e.g., AMD SEV-SNP [3]), but do not describe a full solution; therefore, we do not discuss them any further here. So, in the rest of this section, we focus on the TEEs that provide a complete description of their support for sealing: Flicker [5], SEA [6], IBM-PEF [22], Intel SGX [20], TIMBERV [34], Keystone [35], and Sanctuary [31].

A. Sealing Solutions and Trade-offs

All sealing proposals in the surveyed TEEs closely resemble the original proposal based on the TPM [1]. Flicker [5], SEA [6] and IBM-PEF [22] directly rely on the original sealing mechanism of a TPM. This typically involves generating an asymmetric key pair and using it to encrypt a secret such that it can be decrypted (unsealed) successfully only when the system configuration matches the one at the time of encryption (sealing). The system configuration information used during the sealing and unsealing process with a TPM uses the measurements recorded in the TPM's Platform Configuration Registers (PCRs). Since TPMs have limited storage, a typical way to protect large amounts of data is to generate a symmetric key for bulk data encryption and then, seal that key to a TPM.

There exist many different forms of establishing which enclave can unseal previously sealed data: Some TEEs only allow an enclave with the same measurement and on the same platform with a specific TCB version to unseal the data [3], [31], [34]. Other proposals allow all enclaves signed by the same developer to unseal each other's data [20], [70]. Some cloud TEEs also allow enclaves to come with a migration policy to migrate sealed data to a different host [3].

B. Architectural Support for Sealing

Solutions like OP-TEE [71], Keystone [35], TIMBER-V [34] and Sanctuary [31] provide sealing support through their software TCB. Here, the TCB exposes an interface to create sealing keys for each enclave. No additional hardware or architectural support is required in these cases. Hence, this technique is potentially applicable to TEE designs that have a run-time TCB component that is implemented in software. TPM-based solutions require a platform TPM chip that supports the sealing capability discussed above. Since TPMs are usually off-chip components, and are connected over an unprotected bus, such solutions are not secure against A_{bus}. Solutions like Intel SGX [7], [20] expose special CPU instructions in hardware to enable sealing. More specifically, they include instructions to generate and access sealing keys based on different types of binding (e.g., developer identity, enclave measurement).

Finally, in all cases, the architecture must ensure that only the TCB and the owner enclave has access to the sealing key(s). These protections could be implemented through the isolation mechanisms described in Section V.

VIII. TCB Discussion

This section summarizes the TCB composition of existing TEEs designs. However, it is hard to attribute individual design choices to TCB size or complexity. Besides, although the common metric for TCB size is lines-of-code, accurate numbers are not often available for all TCB components. Therefore, we discuss whether TEEs implement their different architectural components in a completely *immutable* way or whether they include elements that are *mutable*. When a component is mutable, it can be changed post-manufacture and hence, updated during the lifetime of the computing platform which is important to avoid costly product recalls. Such updates of the TCB (also called TCB recovery [73]) are typically reflected in the attestation report of the platform. Mutable components usually are implemented in software and immutable components are implemented in hardware. We note that in some cases, software can be implemented in an immutable manner (e.g., Boot ROM) and hardware components could be mutable (e.g., μcode in certain CPUs).

Table III summarizes the components in existing TEE designs that include a mutable element (labelled as M). If an entry in the table labelled as immutable (labelled I), it means that element cannot be changed or updated post manufacture. We use U to denote the cases where the mutability of the implementation is unclear.

	Name	RTM	Measurement	Attestation	Isolation		Secure IO	Secure storage	TCB size	
					CPU	Memory			Boot	Monitor
Industry	Intel SGX [7], [20], [72]	I	M	M	M	U+M	-	M	NA	NA
	Intel TDX [8]	I	M	M	M	M	-	-	0	NA
	AMD SEV-SNP [3]	I	U	M	U	U	-	U	NA	NA
	ARM TZ [21]	I	M	M	M	M	M	-	50k LoC	50k LoC†
	ARM CCA [9]	I	M	M	M	M	M	-	50k LoC	NA
	IBM PEF [22]	I	M	M	M	M	-	M	400k LoC	75k LoC
Academia	Flicker [5]	I	M	M	M	-	-	I	0	0.25k LoC
	SEA [6]	I	M	M	M	-	-	I	0	NA
	SICE [23]	I	M	M	M	M	-	-	NA	2.1k LoC
	PodArch [24]	-	-	-	I	I	-	-	0	0
	HyperCoffer [25]	-	-	-	I	I	-	-	NA	1.1k LoC
	H-SVM [26], [27]	I	M	M	M	M	-	-	NA	1.4k LoC
	EqualVisor [18]	I	-	-	M	M	-	-	NA	1.2k LoC
	xu-cc15 [28]	-	-	-	I	I	-	-	0	0
	wen-cf13 [29]	-	-	-	I	I	-	-	0	0
	Komodo [30]	I	M	M	M	M	-	-	0.8k LoC	2.7k LoC
	SANCTUARY [31]	I	M	M	M	M	-	M	50k LoC	51.5k LoC†
	TrustICE [32]	-	-	-	M	M	M	-	50k LoC	0.28k LoC
	HA-VMSI [33]	I	M	M	M	M	-	-	NA	3.5k LoC
	Sanctum [4]	I	M	M	M	M	-	-	0.4k LoC	5k LoC
	TIMBER-V [34]	I	M	M	M	M	-	M	NA	2k LoC
	Keystone [35]	I	M	M	M	M	-	M	16.5k LoC	10k LoC
	Penglai [36]	I	M	M	M	M	-	-	16.5k LoC	6.4k LoC
	CURE [37]	I	M	M	M	M	M	-	16.5k LoC	3k LoC
	Iso-X [38]	I	I	I	I	I	-	-	0	0
	HyperWall [39]	I	I	I	I	I	-	-	0	0
	Sancus [40], [41]	I	I	I	-	I	-	-	0	0
	TrustLite [42]	I	M	M	M	M	M	-	NA	NA
	TyTan [43]	I	M	M	M	M	-	M	NA	NA
	XOM [44]	-	-	-	I	I	-	-	0	0
	AEGIS [45]	-	-	-	I	I	-	-	0	0

† Using OP-TEE [71] as a base (around 50k LoC).

TABLE III: TCB comparison for the surveyed TEEs. All individual components are marked to be either mutable (M), immutable (I), unknown (U), or not supported (-).

TCB of Verifiable Launch: The RTM in all TEEs that support one is immutable as expected. In most cases, this RTM is only used to measure the very first (or next in DRTM) software TCB component in the chain of trust which directly or indirectly (through later components in the chain) measures the actual enclave. Very few designs, namely, HyperWall [39], Iso-X [38], and Sancus [41] perform the entire enclave measurement in hardware. All other TEEs allow updating the measurement procedure, e.g., to include system parameters such as if hyperthreading is enabled [3], [74]. Attestation is typically also performed by a mutable part of the software TCB with the same exceptions as the measurement above.

TCB of Run-Time Isolation: Many TEEs use a mutable part of the software TCB to implement secure context switches for CPU isolation and manage memory isolation. Exceptions that implement CPU and memory isolation completely in hardware are PodArch [24], Hypercoffer [25], H-SVM [26], Hyperwall [39], Iso-X [38], XOM [44], Aegis [45] and the works by Xu et. al. [28] and Wen et. al. [29]. In some cases, the split of this functionality between mutable and immutable components remains unclear [2], [75].

TCB of Secure IO: Most of the studied TEEs do not support trusted IO. Exceptions include TrustICE [32], CURE [37] and Trustlite [42] and they all rely on mutable software TCB for trusted IO. We note that there are several proposals that focus on enabling trusted IO with some of the TEEs that do not natively support trusted IO as discussed in Section VI.

TCB of Secure Storage: Of the 31 TEE proposals that we studied, 10 proposals discuss sealing support explicitly. Some of these such as Flicker [5], SEA [6] and IBM PEF [22] rely on the TPM for this, while others such as Sanctuary [31], Timber-V [34], Keystone [35] and TrustLite [42] implement this feature as part of their software TCB. While Intel SGX provides similar functionality [20], it is unclear if or what part of it is mutable.

Overall TCB Size: Usually, most TEE designs and implementations seek to minimize the TCB of the TEE architecture to reduce the risk of it being buggy or vulnerable, and in-theory making them more amendable to formal verification. However, in-practice, it is hard to ignore the advantage of being able to update a TCB component if necessary, instead of relying on the TCB being bug-free. Furthermore, as shown in this paper, many TEEs use similar mechanisms overall irrespective of whether they are implemented in a mutable or immutable component. Given this and the fact that existing implementations vary widely in terms of the features they support, we caution against using mutable TCB sizes to compare TEEs. We only include the mutable TCB complexity measured in terms of lines of code obtained from our survey here for completeness.

IX. Related Work

There have been several studies on TEEs that compare them in terms of the types of security protections that they provide; we summarize these below. We limit the following discussion to survey papers on TEEs and exclude the various papers on a single TEE themselves because the latter set of works are the subject of this paper.

Sabt et. al. are among the first to recognize that there were competing TEE definitions around 2015 and therefore, attempt to arrive at a formal TEE definition. Following that, ARM TrustZone-based TEEs from industry and academia were briefly surveyed using this definition [76]. A more detailed discussion of ARM TrustZone-based TEEs, their various flavors across different ARM processor versions, and software solutions that leverage ARM TrustZone can be found in [77]. A further study focuses on the security limitations of ARM TrustZone-based [78]. The primary focus of all these works is on TEE architectures, systems and attacks involving ARM Trustzone. We note that [77] briefly covers a few other non-ARM TEEs but focuses only on the security properties that they enable and not their architectural details.

There have been similar works that survey security mechanisms and TEEs in the RISC-V ecosystem. In [79], the author surveys hardware and architectural security for RISC-V processors and contrasts them to ARM processors. While this paper makes many good comparisons between the ARM and RISC-V architectures in general (e.g., exception levels) as well as with respect to security-related features (e.g., support for cryptography, ISA extensions), it only mentions that the Keystone architecture [35] is similar to ARM TrustZone but defers any further discussion to future work. More recently, existing RISC-V TEE architectures are summarized in [80] but no comparison to TEEs on different processor architectures is given. Similar limitations apply to previous studies on Intel SGX and its applications [81] as well as security limitations [82], [83]. In contrast to these surveys that focus on ARM or RISC-V, this paper covers TEEs across various various instruction set architectures and compares the different micro-architectural elements that underpin them.

The only previous effort at systematization of TEE architectures among all major processor architectures is [84]. Here, the authors summarize TEEs architectures on four dimensions: mechanisms to ensure integrity of the initial contents of the TEE, memory protection, scope (e.g., per system, processor package, core or thread) and finally, developer access. It discusses the security implications of incomplete hardware abstractions that do not capture implementation aspects of TEEs (e.g., timing of operations, caching, concurrency) and finally, covers different applications of TEEs. The discussion in [84] of micro-architectural support for run-time isolation mechanisms in TEEs is limited to memory protection, and, for example, does not include CPU-state protection or trusted IO support. Furthermore, the discussion on memory protection is also not exhaustive and as detailed as covered in this paper.

Existing literature also includes surveys of TEE architectures (e.g., [85], [86]) as well as security properties (e.g., [87], [88]). Although these works cover TEEs across multiple processor architectures, none of them systematically catalog and analyze the various architectural design decisions underlying these TEE designs and analyze their advantages and disadvantages as we do in this paper.

X. Conclusion

In this paper, we analyzed the underlying design choices of commercial and academic TEEs for four high-level security goals: (i) verifiable launch, (ii) run-time isolation, (iii) secure IO, and (iv) secure storage. Even though these proposals often seem very different at first, many of them share many design decisions but use different names and descriptions for them. We believe our findings can help upcoming TEE proposals weigh the different design decisions and hopefully reduces reinvention in the field.

References

[1] T. C. Group, "TCG specification architecture overview," Trusted Computing Group, Tech. Rep., Aug. 2007, revision 1.4.

[2] V. Costan and S. Devadas, "Intel SGX explained." *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[3] D. Kaplan, J. Powell, and T. Woller, "AMD SEV-SNP: Strengthening VM isolation with integrity protection and more," AMD, Tech. Rep., 2020.

[4] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 857–874.

[5] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008, pp. 315–328.

[6] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri, "How low can you go? recommendations for hardware-supported minimal TCB code execution," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 14–25, 2008.

[7] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *Hardware and Architectural Support for Security and Privacy*, 2013, pp. 1–8.

[8] I. Corporation, "Intel® trust domain extensions (Intel® TDX) module base architecture specification," Intel Corporation, Tech. Rep., Sep. 2021, 348549-001US.

[9] Arm, "Arm® architecture reference manual supplement, the realm management extension (RME), for Armv9-A," document number: ARM DDI 0615.

[10] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An off-chip attack on hardware enclaves via the memory bus," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[11] T. Krachenfels, T. Kiyan, S. Tajik, and J.-P. Seifert, "Automatic extraction of secrets from the transistor jungle using laser-assisted side-channel attacks," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 627–644.

[12] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual international cryptology conference*. Springer, 1999, pp. 388–397.

[13] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[14] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[15] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng, "CIPHERLEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 717–732.

[16] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A commodity obfuscation engine on Intel SGX," in *Network and Distributed System Security Symposium*, 2019.

[17] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 431–446.

[18] L. Deng, Q. Zeng, W. Wang, and Y. Liu, "EqualVisor: Providing memory protection in an untrusted commodity hypervisor," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2014, pp. 300–309.

[19] B. Kauer, "OSLO: Improving the security of trusted computing." in *USENIX Security Symposium*, vol. 24, 2007, p. 173.

[20] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13. Citeseer, 2013, p. 7.

[21] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security - enabling trusted computing in embedded systems," *White paper*, 2004.

[22] G. D. Hunt, R. Pai, M. V. Le, H. Jamjoom, S. Bhattiprolu, R. Boivie, L. Dufour, B. Frey, M. Kapur, K. A. Goldman *et al.*, "Confidential computing for OpenPOWER," in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 294–310.

[23] A. M. Azab, P. Ning, and X. Zhang, "SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 375–388.

[24] S. Shinde, S. Tople, D. Kathayat, and P. Saxena, "Podarch: Protecting legacy applications with a purely hardware TCB," *National University of Singapore, Tech. Rep*, 2015.

[25] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 246–257.

[26] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2011, pp. 272–283.

[27] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng, "H-SVM: Hardware-assisted secure virtual machines under a vulnerable hypervisor," *IEEE Transactions on Computers*, vol. 64, no. 10, pp. 2833–2846, 2015.

[28] L. Xu, J. Lee, S. H. Kim, Q. Zheng, S. Xu, T. Suh, W. W. Ro, and W. Shi, "Architectural protection of application privacy against software and physical attacks in untrusted cloud environment," *IEEE Transactions on Cloud Computing*, vol. 6, no. 2, pp. 478–491, 2015.

[29] Y. Wen, J. Lee, Z. Liu, Q. Zheng, W. Shi, S. Xu, and T. Suh, "Multi-processor architectural support for protecting virtual machine privacy in untrusted cloud environment," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2013, pp. 1–10.

[30] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 287–305.

[31] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with user-space enclaves." in *NDSS*, 2019.

[32] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-assisted isolated computing environments on mobile devices," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 367–378.

[33] M. Zhu, B. Tu, W. Wei, and D. Meng, "HA-VMSI: A lightweight virtual machine isolation approach with commodity hardware for ARM," *ACM SIGPLAN Notices*, vol. 52, no. 7, pp. 242–256, 2017.

[34] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, "TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V." in *NDSS*, 2019.

[35] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[36] E. Feng, X. Lu, D. Du, B. Yang, X. Jiang, Y. Xia, B. Zang, and H. Chen, "Scalable memory protection in the PENGLAI enclave," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021, pp. 275–294.

[37] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with customizable and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1073–1090.

[38] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-X: A flexible architecture for hardware-managed isolated execution," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 190–202.

[39] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 437–450, 2012.

[40] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 479–498.

[41] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for IoT devices," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, pp. 1–33, 2017.

[42] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–14.

[43] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proceedings of the 52nd annual design automation conference*, 2015, pp. 1–6.

[44] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *Acm Sigplan Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[45] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2003, pp. 357–368.

[46] I. Lebedev, K. Hogan, and S. Devadas, "Secure boot and remote attestation in the Sanctum processor," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 46–60.

[47] G. E. Suh, D. Clarke, B. Gasend, M. Van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 339–350.

[48] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 42–56.

[49] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan, "VButton: Practical attestation of user-driven operations in mobile apps," in *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, 2018, pp. 28–40.

[50] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "TruZ-Droid: Integrating TrustZone with mobile operating system," in *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, 2018, pp. 14–27.

[51] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 187–199.

[52] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 616–630.

[53] W. Li, H. Li, H. Chen, and Y. Xia, "AdAttester: Secure online mobile advertisement attestation using TrustZone," in *Proceedings of the 13th annual international conference on mobile systems, applications, and services*, 2015, pp. 75–88.

[54] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, 2014, pp. 1–7.

[55] H. Sun, K. Sun, Y. Wang, and J. Jing, "TrustOTP: Transforming smartphones into secure one-time password tokens," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 976–988.

[56] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity GPUs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 455–468.

[57] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi *et al.*, "Fidelius: Protecting user secrets from compromised browsers," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 264–280.

[58] T. Peters, R. Lal, S. Varadarajan, P. Pappachan, and D. Kotz, "BASTION-SGX: Bluetooth and architectural support for trusted I/O on SGX," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018, pp. 1–9.

[59] A. Dhar, E. Ulqinaku, K. Kostiainen, and S. Capkun, "ProtectIOn: Root-of-trust for IO in compromised platforms," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/protection-root-of-trust-for-io-in-compromised-platforms/

[60] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, L. Zhao, F. Yuan, P. Li, Z. Wang, B. Zhao *et al.*, "Enabling privacy-preserving, compute- and data-intensive computing using heterogeneous trusted execution environment," *arXiv preprint arXiv:1904.04782*, 2019.

[61] Synopsys, "Security modules for standard interfaces," accessed on 2021-11-29. [Online]. Available: https://www.synopsys.com/designware-ip/security-ip/interface-security-modules.html

[62] S. Weiser and M. Werner, "SGXIO: Generic trusted I/O path for Intel SGX," in *Proceedings of the seventh ACM on conference on data and application security and privacy*, 2017, pp. 261–268.

[63] O. Kwon, Y. Kim, J. Huh, and H. Yoon, "ZeroKernel: Secure context-isolated execution on commodity GPUs," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[64] H. Oh, K. Nam, S. Jeon, Y. Cho, and Y. Paek, "MeetGo: A trusted execution environment for remote applications on FPGA," *IEEE Access*, vol. 9, pp. 51 313–51 324, 2021.

[65] M. Zhao, M. Gao, and C. Kozyrakis, "ShEF: Shielded enclaves for cloud FPGAs," *arXiv preprint arXiv:2103.03500*, 2021.

[66] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 681–696.

[67] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel, "Telekine: Secure computing with cloud GPUs," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 817–833.

[68] Z. Wang, F. Zheng, J. Lin, G. Fan, and J. Dong, "SEGIVE: A practical framework of secure GPU execution in virtualization environment," in *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2020, pp. 1–10.

[69] H. Oh, A. Ahmad, S. Park, B. Lee, and Y. Paek, "TrustOre: Side-channel resistant storage for SGX using Intel hybrid CPU-FGPA," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1903–1918.

[70] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel software guard extensions: EPID provisioning and attestation services," Intel Corperation, Tech. Rep., 2016.

[71] TrustedFirmware, "OP-TEE documentation," https://optee.readthedocs.io/en/latest/index.html, accessed: 11.11.2021.

[72] Intel, "Xucode: An innovative technology for implementing complex instruction flows." [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html

[73] ——, "Intel® SGX trusted computing base (TCB) recovery," 2018.

[74] ——, "Attestation service for intel® software guard extensions (Intel® SGX): API documentation," https://api.trustedservices.intel.com/documents/IAS-API-Spec-rev-4.0.pdf, revision 4.1.

[75] R. Buhren, C. Werling, and J.-P. Seifert, "Insecure until proven updated: analyzing AMD SEV's remote attestation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1087–1099.

[76] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: what it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1. IEEE, 2015, pp. 57–64.

[77] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.

[78] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1416–1432.

[79] T. Lu, "A survey on RISC-V security: Hardware and architecture," *arXiv preprint arXiv:2107.04175*, 2021.

[80] G. Dessouky, A.-R. Sadeghi, and E. Stapf, "Enclave computing on RISC-V: A brighter future for security?" in *Workshop on Secure RISC-V Architecture Design (SECRISC-V'20)*, Aug 2020.

[81] W. Zheng, Y. Wu, X. Wu, C. Feng, Y. Sui, X. Luo, and Y. Zhou, "A survey of Intel SGX and its applications," *Frontiers of Computer Science*, vol. 15, no. 3, pp. 1–15, 2021.

[82] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of SGX and countermeasures: A survey," *ACM Computing Surveys (CSUR)*, vol. 54, no. 6, pp. 1–36, 2021.

[83] J. Randmets, "An overview of vulnerabilities and mitigations of Intel SGX applications," Cybernetica AS, Tech. Rep., 2021.

[84] L. Zhao, H. Shuang, S. Xu, W. Huang, R. Cui, P. Bettadpur, and D. Lie, "SoK: Hardware security support for trustworthy execution," *arXiv preprint arXiv:1910.04957*, 2019.

[85] L. Coppolino, S. D'Antonio, G. Mazzeo, and L. Romano, "A comprehensive survey of hardware-assisted security: From the edge to the cloud," *Internet of Things*, vol. 6, p. 100055, 2019.

[86] P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Trusted execution environments: properties, applications, and challenges," *IEEE Security & Privacy*, vol. 18, no. 2, pp. 56–60, 2020.

[87] O. Demigha and R. Larguet, "Hardware-based solutions for trusted cloud computing," *Computers & Security*, p. 102117, 2021.

[88] D. C. G. Valadares, N. C. Will, M. A. Spohn, D. F. de Souza Santos, A. Perkusich, and K. C. Gorgonio, "Trusted execution environments for cloud/fog-based internet of things applications." in *CLOSER*, 2021, pp. 111–121.

[89] I. Corporation, "Intel® trust domain extensions," Intel Corperation, Tech. Rep., Aug. 2020, 343961-002US.

[90] AMD, "Secure encrypted virtualization API version 0.24," AMD, Tech. Rep., 2020, issue Date: April, 2020.

[91] J. T. Mühlberg, J. Noorman, and F. Piessens, "Lightweight and flexible trust assessment modules for the internet of things," in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 503–520.

[92] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The RISC-V instruction set manual volume II: Privileged architecture," *EECS Department, University of California, Berkeley*, 2019.

[93] Intel Corporation, "Intel 64 and IA-32 architectures software developer manuals," Jun 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[94] AMD, "AMD64 architecture programmer's manual: Volumes 1-5," revision 4.03. [Online]. Available: https://www.amd.com/system/files/TechDocs/40332.pdf

[95] "Arm® architecture reference manual: Armv8, for Armv8-A architecture profile."

Appendix

In this section we highlight some implementation details of the surveyed TEEs. For the full details of any given architecture, we refer the reader to the primary references for the specific architectures.

A. Verifiable Launch

1) Measurement: The root of trust for measurement roots a chain of trust. Every component in the chain is typically measured (and optionally verified) before it starts executing. The measurement process for all these components in the chain of trust and the enclave itself is similar. It entails mapping the binary executable of the component being measured to memory and computing one or more cryptographic hashes on that memory. Often, such measurements are built incrementally in compact form as a chain of cryptographic hashes on a page-by-page basis [7], [33], [38]. Once the measurement of a component in the chain of trust is complete, its measurement must be stored securely such that an adversary cannot corrupt it. Some TEEs rely on special platform components such as TPM registers to store measurements [5], [6], [22]. However, since TPMs are off-chip components that may be subject to attacks (e.g., by A_{bus}), some solutions store these measurements in on-chip registers [20], [39]. Furthermore, since

	Name	Adversary						Trusted Access Control Information		
		A_{app}	A_{ssw}	A_{tee}	A_{boot}	A_{per}	A_{bus}	MPU	Page Tables	Extra Metadata
Industry	Intel SGX [7], [20]	SL	SL	STL	SL	STL	C	●	○	●
	Intel TDX [8]	SL	SL	STL	SL	STL	C†	●	●	●
	AMD SEV-SNP [3]	STC	STC	STC	-	STL	C†	○	○	●
	ARM TZ [21]	SL	SL	STL	-	SL	C‡	●	●	○
	ARM CCA [9]	STL	STL	STL	-	SL	C‡	○	●	●
	IBM PEF [22]	SL	SL	STL	-	-	-	●	●	○
Academia	Flicker [5]	TL	TL	-	SL	SL	-	○	○	○
	SEA [6]	TL	TL	-	SL	SL	-	○	○	○
	SICE [23]	SL	SL	SL	-	SL	-	●	○	○
	PodArch [24]	STL	STL	STL	-	SC	-	○	●	●
	HyperCoffer [25]	STC	STC	STC	-	-	C	●	○	●
	H-SVM [26], [27]	STL	STL	STL	-	STL	-	○	●	●
	EqualVisor [18]	SL	SL	STL	-	STL	-	○	●	○
	xu-cc15 [28]	STC	STC	STC	?	STC	C	○	○	●
	wen-cf13 [29]	STL	STL	STL	-	STL	C	●	○	●
	Komodo [30]	SL	SL	STL	-	STL	-	●	●	○
	SANCTUARY [31]	STL	STL	STL	-	STL	-	●	○	○
	TrustICE [32]	STL	STL	STL	-	STL	-	●	○	○
	HA-VMSI [33]	SL	SL	STL	-	STL	-	●	●	●
	Sanctum [4]	STL	STL	STL	-	STL	-	●	○	●
	TIMBER-V [34]	STL	STL	STL	-	STL	-	○	○	●
	Keystone [35]	STL	STL	STL	-	STL	-	●	○	○
	Penglai [36]	STL	STL	STL	-	STL	-	○	●	●
	CURE [37]	STL	STL	STL	-	STL	-	●	○	○
	Iso-X [38]	STL	STL	STL	-	STL	-	○	●	●
	HyperWall [39]	STL	STL	STL	-	STL	-	○	●	●
	Sancus [40], [41]	STL	STL	STL	-	SL	-	●	○	○
	TrustLite [42]	STL	STL	STL	-	-	-	●	○	○
	TyTan [43]	STL	STL	STL	-	-	-	●	○	○
	XOM [44]	STC	STC	STC	-	-	C†	○	○	●
	AEGIS [45]	STC	STC	STC	-	STC	C	○	○	●

† No protection against replay attacks.
‡ Memory encryption is optional but available for sale from the manufacturer.

TABLE IV: The various main memory isolation strategies used against individual adversaries. Note that some TEE designs use multiple strategies against different adversaries. Isolation strategies: cryptographic enforcement (C), logical enforcement (L), temporal partitioning (T), spatial partitioning (S), spatio-temporal partitioning (ST) , not considered (-). The required access control information that needs to be maintained by the TCB of the TEEs is split into three groups: memory protection unit (MPU), page table based, and extra metadata.

TPMs and on-chip solutions can only store a limited number of measurements, more recent works store measurements in memory [3], [8], [20], [38], [46] within the enclave's TCB.

2) Attestation: In general, the basic attestation process is well-understood. First, the enclave's underlying TCB constructs an *attestation report* containing the enclave's measurement and its TCB. Then, the attestation report is integrity protected either through a cryptographic signature or a Message Authentication Code (MAC). Digital signatures typically need additional infrastructure. For example, Intel SGX [70] and Intel TDX [89] require a separate Intel service called Intel Attestation Service (IAS) to verify a report containing the measurements. AMD SEV-SNP [90] signs all device-specific keys with an AMD root key and relies on a generic public key infrastructure. TPM-based attestation schemes also require third-party infrastructure such as a trusted third party to generate attestation keys [1]. In contrast, in local attestation, the underlying TCB typically creates a MAC-based report [70] with a local key. We note that certain attestation proposals [34], [41], [43] rely on using a MAC-based scheme even for remote attestation; Only SANCUS [41], [91] discusses how the verifier obtains the symmetric key for verification.

Historically, the only values included in an attestation report were the actual measurements of the enclave and the TCB [5], [6] as well a nonce for freshness. However, more modern TEEs use extended attestation reports to include run-time attributes. For example, Intel SGX [74] and AMD SEV-SNP [3] include a flag in the attestation report to indicate whether simultaneous multithreading (SMT) is enabled. Other types of information that could be added to an attestation report include software version numbers [8], [70], [90], migration policies [90], and TCB version when applicable. Finally, some TEE designs allow the enclave to append some custom data (e.g., a public key certificate) into an attestation report which can be used later to establish a secure channel.

Most TEE designs include a basic set of primitives to support attestation, regardless of whether it is local or remote and independent of the types of information included in the report itself. First, TEE solutions must be able to securely generate and store attestation keys and protect them against unauthorized access/modification. The exact support required for the generation and storage of attestation keys varies based on the key hierarchy a given TEE solution adopts. Most signature-based attestation solutions include a *platform identity key* that is used to establish the authenticity of the platform and one or more *attestation keys* that are signed using the identity key. While attestation keys are usually generated on demand, the platform identity key is either generated once and stored permanently (e.g., Endorsement key in TPM based protocols [1]) or derived on every boot (e.g., based on Root Provisioning key in Intel SGX [70]). Furthermore, a hierarchy of attestation keys could be used to reflect updates to the mutable part(s) of the TCB in the attestation [8], [46], [70], [90]. Second, the TCB in TEE architectures may expose interfaces for requesting (and verifying) an attestation report. These interfaces may be exposed as new instructions (e.g., [7], [8], [38]) or services (e.g., [4], [7], [42], [43]) when implemented in hardware or software respectively.

B. Memory Isolation

Memory Protection Unit (MPU) While many different variants of memory protection units (MPU) exist, they all perform very similar tasks: they check the physical address and the access type (e.g., write or read) against access control information. Most MPU implementations can protect a limited number of memory regions at any point in time and are suitable for coarse-grained memory protections. In order to enable logical isolation, the access control information for the MPU must be configured and controlled solely by the enclave's TCB. Many modern academic TEEs rely on such an MPU to provide isolation [4], [31], [34], [35], [37].

Memory Management Unit (MMU) In contrast to the coarse-grained MPU, MMUs allow for much finer-grained access control checks. An MMU is a logic block that converts a virtual address to a physical address using data structures called page tables. Page tables not only store these mappings but also additional security-sensitive information such as permissions (e.g., read, write, execute) alongside every entry. In TEE designs, MMUs can be used for logical memory isolation and offer better scalability and more fine-grained protections compared to MPU based approaches. However, the page tables themselves must be protected from the adversary. There are a variety of ways to manage the page tables: The most straightforward way to manage page tables is by letting the TCB control all the page tables not only for all the enclaves but also all for the other untrusted software components (e.g., A_{ssw}, A_{app}) [18], [25], [33], [33]. In other proposals, the TCB controls only the page tables of all enclaves; so, the TCB can ensure that the enclave has access only to its own memory and prevent it from accessing the memory of other enclaves. However, if the TCB does not control the page tables of A_{ssw}, it cannot prevent such a privileged attacker from accessing an enclave's memory. Examples of architectures that do this include Intel TDX [8], ARM TZ [21], and ARM Realms [9]. In such designs, the MMU must use a set of *secondary metadata* to enable access to enclave memory based on the executing context (e.g., as recognized by the CPU mode or privilege level or other identifiers). This approach based on secondary metadata can also be used when the system design does not provide a way to set up trustworthy page tables for an enclave itself [3], [4], [7], [24], [38].

Translation Look-aside Buffers (TLBs): A translation look-aside buffer (TLB) essentially holds information about the MMU's recent virtual to physical translations. From a security perspective, it is essential to prevent the re-use of stale MMU translations by untrusted execution contexts. In modern processors, dedicated TLBs are usually available per thread of execution (e.g., per logical processor) and are hence, inherently spatially partitioned among concurrent threads. Furthermore, when a given logical processor is shared over time by different execution contexts, usually temporal isolation is used, i.e., TLBs are flushed on every context switch. However, since TLB flushes can degrade performance, modern processors support the use of additional information about the execution context that owns each entry to enable partial TLB flushes during transitions [92]–[95]. This allows multiple execution contexts to share the TLB as a whole at any given point in time but prevents them from re-using each other's translations. Besides dedicated TLBs per logical processor and TLB flushes on context switches, some solutions [3], [8] rely on additional tags in their TLBs to protect translations belonging to enclaves. Lastly, we did not see the usage of cryptographic isolation for TLBs, probably due to its potential cost and performance overheads.

Cryptography Support in the Memory Controller: While it is possible to include simple access control checks into the memory controller, the more common type of memory isolation enforcement at the memory controller is cryptographic. Encryption, integrity, and replay protection are are applied to data in the memory controller just before it exits the SoC. When used for protection against a A_{bus}, this isolation technique is typically implemented using a single pair of keys - one for encryption and the other for integrity protection (e.g., Intel SGX). Sometimes, critical ranges of memory are assigned separate keys as a defense-in-depth mechanism, but overall, defense against A_{bus} usually requires just a handful of keys (e.g., ARM CCA which uses a separate key per world). Metadata related to integrity and anti-replay protection are often stored in sequestered memory; the metadata corresponding to each memory transaction is retrieved/stored by memory subsystem usually in a way that is software-agnostic/entirely hardware managed (e.g., Intel SGX). The underlying micro-architectural components include a block that can generate these additional accesses to sequestered memory, a metadata cache for recently accessed memory, and filters to ensure only authorized access to this metadata.

When cryptographic enforcement is used to protect against software attacks (e.g., A_{tee}, A_{ssw}) and malicious peripherals

(A_{per}), TEE architectures use a separate key per execution context. They commonly use a unique key per enclave that is not available for use by other software adversaries or peripherals. Such solutions also require that information about the source of the memory request is carried along with every transaction all the way from the CPU to the memory controller. The memory controller uses this information to select the correct cryptographic key material for accessing the actual memory targeted by that transaction. This information about the execution context corresponding to the memory transaction is often retained in the TLBs and caches, and it is carried on all system fabrics to enforce access control within the SoC.