IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 2 May 2025, accepted 11 June 2025, date of publication 19 June 2025, date of current version 2 July 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3581487

RESEARCH ARTICLE

AnyTEE: An Open and Interoperable Software Defined TEE Framework

DAVID CERDEIRA[1], JOSÉ MARTINS[1], NUNO SANTOS[2], (Member, IEEE), AND SANDRO PINTO[1]

[1]ALGORITMI Research Centre/LASI, Universidade do Minho, 4800-058 Guimaraes, Portugal
[2]INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, 1049-001 Lisbon, Portugal

Corresponding author: David Cerdeira (david.cerdeira@dei.uminho.pt)

This work was supported in part by the ALGORITMI Research Centre/LASI at the Universidade do Minho, the Instituto Técnico/Universidade de Lisboa, and the Fundação para a Ciência e a Tecnologia (FCT) under Project UIDB/50021/2020 and Project UIDB/00319/2020; and in part by European Union's Horizon Europe Research and Innovation Program for the Project Cross-Platform Open Security Stack for Connected Devices (CROSSCON) under Grant 101070537. The work of David Cerdeira and José Martins was supported by FCT under Grant SFRH/BD/146231/2019 and Grant SFRH/BD/138660/2018.

ABSTRACT Trusted Execution Environments (TEEs) are fundamental in securing a broad spectrum of applications and are prevalent on various platforms, from mobile and embedded devices to cloud servers. However, the existing landscape of TEE technologies is notably heterogeneous, posing substantial interoperability and compatibility challenges for application developers and system designers. These hurdles are not limited to well-established technologies like Arm TrustZone and Intel SGX, but also extend to newer generations of TEE models. In this context, we introduce AnyTEE, a new framework that allows application developers to use multiple TEE models in their applications and system designers to engineer customized TEE models across multiple Instruction Set Architectures (ISAs). Essentially, AnyTEE harnesses widely available hardware virtualization primitives to facilitate the development of Software Defined TEEs (sdTEEs). Our implementation of AnyTEE includes sdTEE versions of the SGX and TrustZone models for Arm and RISC-V, respectively, along with an sdTEE variant of SGX that incorporates fine-grained isolation mechanisms. Our evaluation shows that AnyTEE achieves near-native performance (<3% overhead).

INDEX TERMS Trusted execution environments, virtualization, TrustZone, intel SGX, system security.

I. INTRODUCTION

Trusted Execution Environments (TEEs) are a pivotal and continuously evolving technology for protecting security-sensitive software. Available on most Commercial Off-The-Shelf (COTS) platforms, TEEs rely on hardware-based primitives to isolate the execution of said software from the untrusted main Operating System (OS) (or hypervisor). Well-established TEE technologies from major CPU vendors include Intel SGX [17], Arm TrustZone (TZ) [51], and AMD SEV [2], and target a wide spectrum of applications ranging from cloud servers and desktops, to mobile and embedded / Internet of Things (IoT) devices [17], [51], [56]. Further, steamed by the ever-growing push for confidential computing, new TEE-enabling technologies have recently emerged, such as Intel TDX [30], Arm CCA [38], and RISC-V CoVE [55].

However, system designers and integrators are today hampered by significant interoperability and compatibility challenges due to proprietary features and peculiarities of TEE technologies. In fact, to cope with stronger threat models and deliver richer functionality while providing a competitive market advantage, TEE vendors have developed independent solutions, resulting in heterogeneous TEE enforcement mechanisms and protection models. For example, while TZ mirrors the CPU privilege levels, Intel SGX preserves the same privilege level while providing additional protections that prevent arbitrary access to the TEE by system-level software. These differences translate into separate software development niches for TZ and SGX, preventing software reusability between these communities and forcing developers to be proficient in both

The associate editor coordinating the review of this manuscript and approving it for publication was Diana Gratiela Berbecaru.

© 2025 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

TEE-enabling technologies. Unfortunately, this fragmentation of TEE software development niches is expected to grow as the next generation of TEE hardware also exhibits its specific traits. For instance, while SGX and TZ offer fine-grained protection at the application logic, Intel TDX and Arm CCA enable coarser isolation units spanning entire Virtual Machines (VM).

To cope with TEE heterogeneity, researchers have proposed to emulate specific TEE technologies on platforms that do not natively support them [20], [28], [33], [66]. For example, while Komodo [20] mimics the SGX protection model on Arm platforms, HyperEnclave [33] provides SGX compatibility on AMD servers. These works, however, do not allow the coexistence of different TEE programming models (e.g., emulated SGX and TZ environments) nor do they allow the development of novel or customizable TEEs. Another approach to dealing with TEE heterogeneity consists of developing elementary hardware abstractions for building customizable TEEs. Keystone [35] pioneered this concept by providing such abstractions atop raw hardware security primitives, i.e., the RISC-V physical memory protection (PMP). The PMP is often present in RISC-V computers (is part of the ISA) and is a per-CPU protection unit with a limited number of configurable memory regions (typically 16). However, since PMP is only available in RISC-V, Keystone cannot directly work on other ISAs, hampering the porting effort cross-platforms. Furthermore, Keystone does not provide seamless compatibility with existing TEE programming models, e.g., precluding the integration of legacy TZ applications.

In this paper, we complement these existing approaches and make the case for *Software Defined TEEs* (sdTEEs), i.e., a TEE emulation and customization paradigm that simultaneously offers three fundamental properties: (1) *portability*, (2) *configurability*, and (3) *interoperability*. Firstly, it should be possible to run legacy stacks (binaries, applications, and APIs) across various platforms and architectures. Secondly, they should provide extensible primitives and building blocks, enabling developers to implement their own custom TEEs tailored to specific requirements. Finally, Software Defined TEEs should provide interoperability among TEE solutions (e.g., application- and VM-level), while enabling developers to use multiple TEE models simultaneously. To the best of our knowledge, no existing solution has achieved this so far.

To materialize this concept, we propose AnyTEE, an open-source framework that leverages the widely available hardware virtualization primitives existing in well-established computing platforms (e.g., Arm VE, Intel-VT, RISC-V Hypervisor extension) to create sdTEEs. sdTEEs consist of (virtual) isolated execution environments linked to an associated context and set of resources (e.g., memory, interrupts) that can be hierarchically related. To this end, AnyTEE introduces the concept of *hierarchical TEE execution modeling*, i.e., a *parent* sdTEE oversees a *child* sdTEE, facilitating, for example, the ability to mimic the vanilla TZ model, where the secure world has higher privileges than, and control of, the normal world. This design enables composability, nesting, and/or customization, to support legacy, modern, and future TEE models. TEE models for confidential computing can be replicated by resorting to nested virtualization techniques, and new sdTEEs featuring additional fine-grained security properties (e.g., intra-privilege isolation) by manipulating page tables.

We developed a reference implementation of AnyTEE for (i) the SGX protection model (sdSGX) on a COTS Arm platform (NXP i.MX8QM) and for (ii) the TZ protection model (sdTZ) on a RISC-V SoC running on an FPGA. In both cases, we run legacy trusted applications (TAs) without modifying the source code, e.g., we seamlessly run a TZ Bitcoin wallet TA on a RISC-V machine. We also support the simultaneous execution of sdTZ and sdSGX TEE models. Furthermore, our implementation demonstrates how to augment the SGX and TZ protection models, for example, by restricting the memory access privileges of the trusted OS running in the sdTZ [13]. We also implement fine-grained security enhancements by enhancing sdSGX enclaves with rapid permission context-switching, akin to VMFUNC [46]. We evaluated AnyTEE using microbenchmarks and legacy real-world applications. Our evaluation shows that, in general, our system achieves near-native performance (average less than 3%), mainly due to memory virtualization translation at the hardware level. Our work is open source [8].

A. CONTRIBUTIONS

We make the following contributions:

- Conceptualization of *Software Defined TEEs* (sdTEEs) with a focus on three main properties: portability, interoperability, and configurability.
- *AnyTEE framework design*, outlining the building blocks that enable the support of existing and novel TEE models on platforms with hardware-supported virtualization.
- *Open-source reference implementation* for SGX (sdSGX) and TZ (sdTZ) protection models, accommodating diverse ISAs such as Armv8-A and RISC-V. Additionally, an sdTEE-based intra-environment isolation mechanism.
- Comprehensive *evaluation and benchmarking of AnyTEE* across SGX and TZ TEE models, and *theoretical security analysis*, highlighting mitigations of potential attacks vectors.

II. BACKGROUND AND OVERVIEW

TEE technologies vary widely in their architectures, capabilities, and threat models. In this section, we explore key issues that arise from this heterogeneity. First, we discuss the diversity of TEE technologies. Then, we examine the limitations of both TEE Software Development Kits (SDKs) and existing solutions for decoupling TEE software from hardware. Finally, we outline this work's goals.

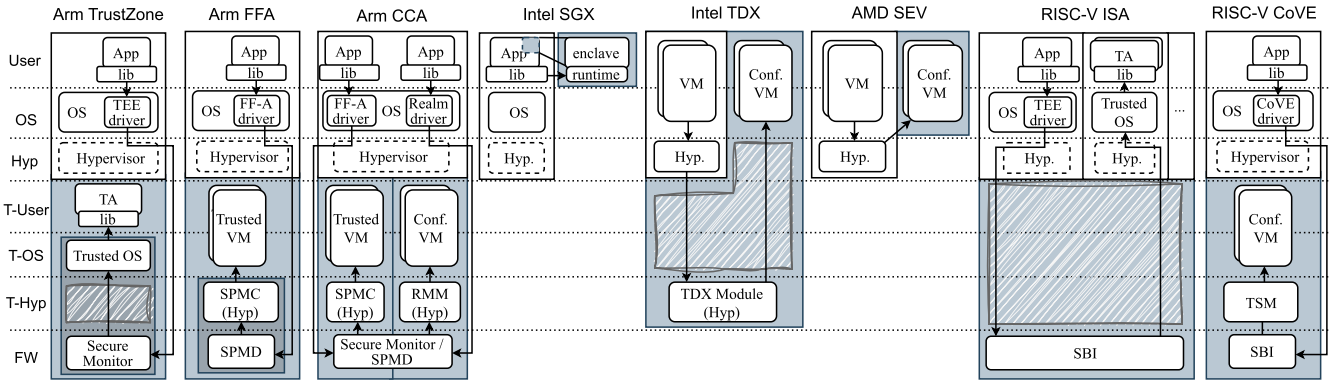


FIGURE 1. Software stacks and privilege levels of various TEE technologies. Inspired by related work [12], we depict privileges vertically and in ascending order, with the lowest privilege at the top. TZ: the OS in the normal world requests services to the TEE in the secure world through the secure monitor. FFA: a trusted hypervisor is added to manage trusted VMs. CCA: the Realm world is added to host the Realm Manager (RMM) which manages confidential VMs. SGX: enclaves are tied to a process and are protected from all other system software. TDX and CoVE: a trusted hypervisor creates confidential VMs with resources donated by the untrusted Hypervisor and performs context switching. SEV: an untrusted hypervisor manages VMs, which can either start up fully protected or select which memory is protected. RISC-V ISA: the physical memory protection (PMP) is used to create security domains, with SBI managing context switching.

A. HETEROGENEITY OF TEE TECHNOLOGIES

A common denominator between existing TEE technologies lies in their ability to protect the execution state of security-sensitive software, isolating it from an untrusted host OS or hypervisor. Yet, they possess unique features that make them incompatible with each other. To better understand their primary sources of incompatibility, in this section, we briefly review representative TEE technologies.

1) REPRESENTATIVE TEE TECHNOLOGIES

As illustrated in Figure 1, major hardware manufacturers have developed various TEE technologies. Arm is one of the major players, creating TZ [51], which is widely deployed in numerous mobile and embedded/IoT devices. Evolving from TZ, Arm Firmware Framework for Arm (FFA) [4] and Confidential Computing Architecture (CCA) [38] ensued. From Intel, we find Software Guard Extensions (SGX) [17] for desktop and server platforms, and Trust Domain Extensions (TDX) [30] for cloud and data center environments. AMD engineered their own TEE technology for the cloud, namely Secure Encrypted Virtualization (SEV) [2]. Importantly, RISC-V also includes primitives for building TEE software stacks in its baseline specification (ISA) [21] and is now standardizing the Confidential Virtual Machine Extension (CoVE) [30].

2) DIVERSITY OF TEE PROTECTION MODELS

Conceptually, TEE technologies offer different protection abstractions, which are then reflected in the programming and usage models presented to software developers. One approach aims to protect a specific piece of application logic, typically a *trusted function*, that provides a well-defined interface for invocation by the unprotected application component. This trusted function is referred to as a TA in TZ or RISC-V ISA, or as an enclave program in Intel SGX. This model is particularly well-suited for desktops, mobile devices, and low-end hardware. A second approach offers a coarser level of protection by enabling the isolation of an entire VM, commonly known as a *confidential VM*. As illustrated in Figure 1, confidential VMs are supported by Arm FFA, Arm CCA, Intel TDX, AMD SEV, and RISC-V CoVE. This model is crucial to enabling confidential computing.

3) DIVERSITY OF TEE ENFORCEMENT MECHANISMS

For protecting TEE memory and execution state, TEE technologies typically leverage CPU privilege levels. One way consists of introducing *new trusted CPU privilege levels* to protect the TEE. For instance, TZ mirrors the CPU privilege levels of the untrusted environment, where the OS or hypervisor runs, into a trusted environment where TAs execute. The former is called the normal world, and the latter, the secure world. The OS must perform a secure context switch to transition from the normal world, execute the TA with user-level privileges in the secure world, and then transition back to the normal world once the TA execution finishes. Two trusted software components are necessary to facilitate this process, both hosted in the secure world: the secure monitor and the trusted OS. Subsequent Arm TEE technologies largely build upon the trusted CPU privilege levels introduced by Arm TZ [13]. In contrast, other TEE technologies *preserve the same CPU privilege level layout but enhance it with additional protections* that prevent arbitrary access to the TEE by system-level software. Notably, Intel SGX allows the creation of user-level TEE domains, called enclaves, which prevent the host OS or hypervisor from accessing their internal state, although they retain their privilege to manage the memory allocation of the enclaves. Only a small trusted library needs to be linked as part of the enclave program to enable context switching. AMD SEV

TABLE 1. Comparison of SDKs based on scope, support for unmodified TA support, configurable security models, and platform and system support.

	SDK Scope	Support for unmodified TAs	Configurable Security Model	Platform / System Support	
				TEE	Other
Graphene-SGX [61]	TA	✓	✗	SGX	✗
Scone [6]	TA	✓	✗	SGX	✗
Asylo [7]	TA	✓	✗	SGX	✗
Enarx [19]	TA	✗	✗	SGX, SEV	✗
SOFAEnclave [58]	TA	✓	✗	SGX, SEV	✗
secGear [48]	TA	✓	✗	SGX, TZ(iTrustee)	✗
MesaTEE [60]	TA	✗	✗	SGX, TZ(OP-TEE)	✗
Open Enclave SDK [57]	TA	✗	✗	SGX, TZ(OP-TEE)	✗
UniTEE [24]	TA	✗	✗	SGX, SEV, TZ(UniTEE microkernel)	✗
Veracruz [16]	TA	✗	✗	SGX, TZ(OP-TEE), Nitro Enclave	IceCap Hypervisor

adopts a similar approach to protect confidential VMs at the OS privilege level.

4) OTHER TEE FEATURES

In addition to isolated execution, TEEs tend to include two additional features: (1) *verifiable launch*, which involves the ability to generate evidence that the TEE is running unaltered software through the use of trusted boot and remote attestation protocols; and (2) *trusted I/O*, enabling TEEs to access external trusted devices, including secure storage, random number generators, cryptographic engines, and more. Memory encryption is an optional feature for protecting against physical attacks, such as cold boot attacks or bus snooping. It is especially relevant in cloud deployments, hence its adoption into SGX, SEV, and TDX. TZ-based platforms typically lack encrypted memory. For an in-depth analysis, we refer the reader to Schneider et al. [56].

B. TEE SDK LIMITATIONS

TEE SDKs aim to provide compatibility and portability. However, we argue that SDKs achieve these goals only in a limited way. To support our view, we examine ten widely used TEE SDKs, evaluating their scope, compatibility with existing software, support for platforms without TEE hardware, support for flexible threat models, and target ISAs. A detailed discussion is presented below. Table 1 summarizes the analysis.

1) SDK SCOPE

TEE SDKs are primarily designed to help developers create TAs by offering APIs that abstract the complexities of interacting with TEE hardware and software. However, they fall short of enabling the seamless deployment of larger software stacks, such as a trusted OS (e.g., TZ TEE trusted OS) or a hypervisor (e.g., TZ TEE trusted Hypervisor), across different platforms. This limitation undermines the portability and scalability of complex, system-level trusted environments, for example, a full TZ stack involving a trusted hypervisor and multiple trusted OSes, or a confidential computing stack with an untrusted hypervisor.

2) SUPPORT FOR UNMODIFIED TAS

SDKs often require developers to modify their code to comply with specific APIs. For example, a TA written for Google's Trusty trusted OS, or OP-TEE, would have to be rewritten to target a different SDK. Some SDK solutions like Graphene-SGX [61] and Scone [6] offer support for unmodified applications. However, this support is limited to Linux applications and does not extend to legacy TAs (e.g., TZ TEE apps or SGX enclaves).

3) CONFIGURABLE SECURITY MODEL

No TEE SDK currently allows developers to customize the security model. These SDKs offer predefined models and permissions that may not address all use cases or threat scenarios. For instance, TZ TEEs often grant unrestricted access to normal world resources, creating potential attack vectors [12], and SDKs lack the necessary mechanisms to enforce stricter access control. In contrast, SGX enclaves face different limitations. For example, enclaves rely on the untrusted app and the OS to access IO devices. While SDKs assist in managing IO operations, they cannot provide direct device access to mitigate high overheads caused by IO-intensive SGX enclave operations.

4) TEE SYSTEM SUPPORT

SDKs often support a very limited number of TEEs. Asylo [7], Scone [6] and Graphene-SGX [61] focus on Intel SGX. Enarx [19] and SOFAEnclave [58] support Intel SGX and AMD SEV but none of the Arm or RISC-V TEEs, restricting their use in mobile or IoT contexts. Open Enclave SDK [57], MesaTEE [60], and secGear [48] also offer limited TEE support, only supporting Intel SGX and TZ, with OP-TEE support in Open Enclave SDK and MesaTEE and iTrustee for secGear. In contrast, Veracruz [16], and UniTEE [24] provide broader architecture support. Veracruz [16] supports Intel SGX, TZ's OP-TEE, and AWS Nitro Enclaves. UniTEE supports Intel SGX, AMD SEV, and TZ with a custom microkernel. Regarding TZ, SDKs generally offer compatibility with Global Platform APIs, as seen in OpenEnclave SDK, secGear, Veracruz, and MesaTEE. Although the Global Platform APIs aim to standardize TEE

interfaces across vendors, they are not universally supported. For instance, Google's Trusty does not implement them.

5) NON-TEE SYSTEM SUPPORT

While hardware TEE primitives are widespread in well-established ISAs and modern platforms, there are popular platforms, such as the Raspberry Pi, that do not feature full TEE hardware support. For example, the RPi4 SoC (BCM2711) supports TZ at the CPU level (privilege levels and security states) but lacks key platform-level TZ controllers, i.e., TZASC, to protect secure memory. SDKs do not address this problem as they rely on the TEE hardware or on TEE software below them to offer the appropriate isolation mechanisms. Veracruz [16] stands out as an exception, as it supports executing its runtime atop the IceCap hypervisor.

In summary, while TEE SDKs aim to enhance compatibility and portability, their effectiveness is limited. These SDKs generally fall short in supporting unmodified TAs, offering configurable security models, and broad system support, particularly for non-TEE hardware platforms. Moreover, SDKs don't provide solutions for the integration of complex trusted systems, or to adapt to different threat models, which further restricts their practical utility.

C. DECOUPLING TEE SOFTWARE FROM HARDWARE

Traditional TEEs are tightly coupled to specific hardware implementations, limiting portability and flexibility. Recent research has focused on decoupling TEE software from hardware, enabling trusted software to run independently of specific hardware-based TEE implementations. There are two main classes of approaches currently proposing solutions to decouple TEE Software from TEE Hardware. The first is to allow platform-specific TEE software to execute on other platforms, as demonstrated by solutions like HyperEnclave [33] and MyTEE [28]. While MyTEE relies only on virtualization primitives, HyperEnclave leverages both virtualization and TEE primitives to achieve the desired security properties. The second approach, introduced by Lee et al. with Keystone [35], offers configurable security properties at design time, allowing developers to tailor the threat model to specific use cases. However, existing solutions face two main limitations. First, they do not support TEE software from multiple TEE technologies due to rigid execution models, e.g., HyperEnclave cannot host TZ TEE software, and MyTEE cannot support SGX enclaves. Second, ensuring compatibility with existing TEE software remains challenging when offering configurable TEE properties, as demonstrated by the limitations of Keystone. Keystone is restricted by the PMP mechanism, e.g., a limited number of regions cannot address arbitrary dynamic allocation, and the reliance on custom runtimes to host TAs, offering no compatibility with existing TEE software. We provide a detailed analysis of the limitations of these systems in §VIII-A.

Although decoupling TEE software from hardware offers significant advantages in flexibility, it can lead to the loss of some security properties. For instance, an SGX enclave that depends on memory encryption to protect sensitive data cannot be securely deployed on platforms without such a mechanism. However, in the absence of necessary hardware, it may be possible to implement these security features through software-based solutions, in this case, a software-based memory encryption solution.

D. ANYTEE GOALS

With AnyTEE, we address the portability and configurability limitations of both TEE SDKs and TEE decoupling solutions. Our approach relies on hardware-assisted virtualization, a widely accessible technology even on low-cost platforms (e.g., Raspberry Pi), to isolate TEE software and emulate low-level TEE hardware features, ensuring compatibility with existing TEE implementations. Additionally, we offer a flexible and extensible framework that allows developers to customize the TEE to meet specific security requirements, providing a tailored solution that adapts to diverse threat models and use cases. project requires developers to consider platform limits when emulating TEE models. For example, missing memory encryption weakens physical attack defenses. In summary, AnyTEE aims to achieve three main goals:

1) PORTABILITY

Ease of transitioning TEE software stacks between various platforms and architectures, including those lacking native TEE support.

2) CONFIGURABILITY

Allow applications to benefit from custom TEEs or modify the security properties of existing ones to meet specific needs.

3) INTEROPERABILITY

Allow applications to benefit from different TEE models co-existing simultaneously on the same platform.

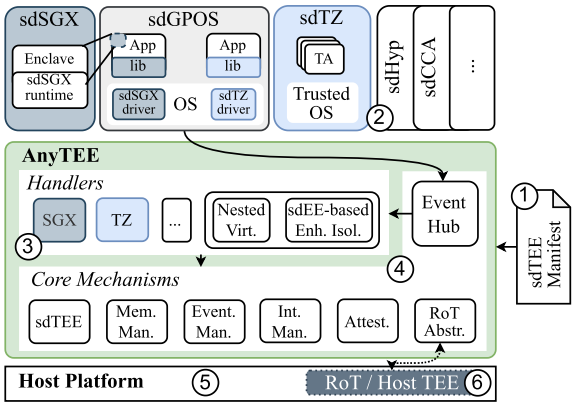


FIGURE 2. AnyTEE architecture, with multiple sdTEE types.

III. DESIGN

AnyTEE is our proposed solution to offer software defined TEEs compatible with existing TEE software on commodity platforms. In this section, we present the design of AnyTEE,

its components, and the approach to create cross-platform, legacy-compatible, and configurable TEEs. Figure 2 presents AnyTEE architecture and its software defined TEE interface. In our system, software defined TEEs, referred to as sdTEEs, are defined through an *sdTEE Manifest* ① and TEE software ②. sdTEEs are managed by AnyTEE *Handlers* ③, which use the internal core mechanisms ④ as building blocks. AnyTEE executes on a host platform ⑤, that may feature its own host TEE or a Root-of-Trust ⑥, which AnyTEE leverages to provide features such as monotonic counters, secure storage, and true random numbers.

In AnyTEE, an attacker's goal is to violate the access control policy, either to compromise sdTEEs or other parts of the system. We assume AnyTEE implementation is correct and that it is securely integrated into the platform's boot process, which includes secure loading of AnyTEE and sdTEE Manifest. The sdTEE software stacks are not trusted. Side-channel, micro-architectural, or physical attacks are out of our scope, as mitigations to these attacks are orthogonal to our work.

A. SOFTWARE DEFINED TEES (SDTEES)

Through sdTEE configuration and management, AnyTEE allows the creation of TEE programming models. Handlers oversee sdTEE execution and integrate into AnyTEE as plugins written in the C programming language that handle sdTEE requests and events implementing TEE functionality.

1) SDTEE ACCESS CONTROL AND STATE

sdTEEs comprise an execution context with access to memory (including MMIO), interrupts (configured through the interrupt controller), and execution priority relative to other sdTEEs. sdTEEs execution context is a VM with multiple virtual CPUs (vCPUs), one per physical CPU. The execution context includes CPU registers, system registers, nested page tables, and a virtual interrupt control state. Nested page tables are used to enforce sdTEE's permissions over the platform's memory and MMIO, enabling granular access control. sdTEEs may need access to interrupts to respond to asynchronous events, such as timer and asynchronous device events, and in these cases, sdTEE's state includes a virtual interrupt controller. sdTEE execution priority determines execution precedence among sdTEEs. For instance, in TZ, the secure world can preempt normal world execution by setting interrupts as secure, resulting in higher priorities interrupt handling. On the other hand, in SGX, enclave execution priority is lower than the GPOS's, allowing the GPOS to take control of the enclave.

2) HIERARCHICAL MODELING OF TEE EXECUTION

AnyTEE introduces a hierarchical modeling approach for TEEs based on an execution control hierarchy. This approach categorizes controlling partitions as *parents*, while controlled partitions are identified as *children*. In TZ, for instance, the secure world serves as the parent of the normal world. This approach allows for the concurrent execution of multiple TEEs, exemplified in Figure 3 (A), where sdSGX is controlled by sdSGPOS, which is, in turn, controlled by sdTZ. Furthermore, our modeling approach supports nested TEEs, as depicted in Figure 3 (B). This enhances system compartmentalization and security by adhering to the least privilege principle. For example, it enables the execution of a TEE with exclusive resource access that sdTZ cannot reach. The hierarchical view of TEEs also allows a reconsideration of the TEE execution model, as shown in Figure 3 (C). This facilitates novel configurations, such as a system running a TZ environment under sdGPOS control, preventing sdTZ from taking control of sdGPOS execution thus improving sdGPOS availability. Additionally, it can be used to enhance the properties of sdSGX enclaves, for example, to ensure minimum enclave execution times.

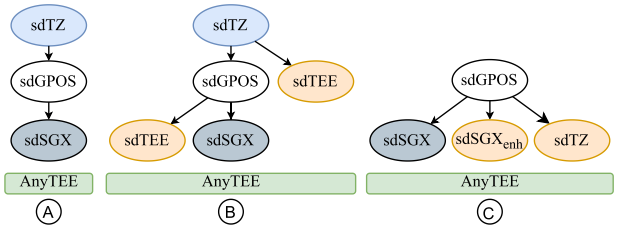


FIGURE 3. Hierarchical sdTEEs enable TEE composability (A), nesting (B), and customization (C).

3) SDTEES CONFIGURATION

The configuration of each sdTEE access to platform memory, MMIO, interrupts, and its position in the execution hierarchy is done through a file named the sdTEE Manifest. For instance, establishing a TZ TEE involves configuring the manifest with two instantiated sdTEEs: one for the normal world (sdGPOS) and one for the secure world (sdTZ). The sdTZ manifest is defined first, modeling sdTZ as the parent of sdGPOS. Using the partition field, sdTZ is granted access to all memory, MMIO, and interrupts except for AnyTEE's private memory. The trusted OS binary is set as the manifest payload. The main handler is the sdTZ handler. Subsequently, the sdGPOS manifest is created as sdTZ's child. sdGPOS's partition fully overlaps with sdTZ but excludes resources exclusive to sdTZ. The payload for sdGPOS is set as the normal world bootloader (e.g., u-boot), responsible for booting the GPOS. The main handler for sdGPOS is the sdGPOS handler, and sdGPOS can make requests to the sdTZ handler.

B. SUPPORT FOR TEE MODELS

AnyTEE employs a modular approach to emulate TEEs, by introducing *sdTEE Handlers*. All sdTEEs require a main handler to deliver interrupts and handle events according to TEE-specific logic. Handlers integrate with AnyTEE by using AnyTEE's core mechanisms and event-based system, similar to XMHF [64]. During sdTEE setup, handlers register callbacks to events for sdTEE instances. When events occur

TABLE 2. AnyTEE main API for handlers.

Module	API	Mechanisms	Module	API	Mechanisms
sdTEE	`create`	Create an sdTEE using an sdTEE manifest	Mem. Man.	`add_mem_region`	Add memory region to an sdTEE dynamically
sdTEE	destroy	Destroy an sdTEE	Mem. Man.	`rm_mem_region`	Remove memory region from an sdTEE dynamically
Event	`sub_fw`	register callback on sdTEE firmware calls	Mem. Man.	`add_dev_region`	Add MMIO region to an sdTEE dynamically
Event	`sub_hvc`	register callback on sdTEE hypervisor calls	Mem. Man.	`rm_dev_region`	Remove MMIO region from an sdTEE dynamically
Event	`sub_abort`	register callback on sdTEE access faults	Mem. Man.	`mem_translate`	Translate an sdTEE address to a physical address
Event	`sub_mem_region`	register callback on sdTEE access to region	Int. Man.	`inject_interrupt`	Inject interrupt into sdTEE vCPU
Event	`sub_interrupts`	register callback on sdTEE interrupt events	Exec. Man.	`push`	Add sdTEE vCPU to the execution stack
Attest.	`get_quote`	Obtain an sdTEE attestation quote	Exec. Man.	`pop`	Remove sdTEE vCPU from the execution stack

or a request is made for an sdTEE instance, AnyTEE iterates over a list of the registered callbacks and invokes them iteratively allowing each handler to react appropriately. The sdTEE manifest field *accessible handlers* controls which requests are allowed for each sdTEE. Handlers can register to manage events by using the `sub_`$\star$ AnyTEE functions, see *AnyTEE Core Mechanisms* below.

1) SDTEE EXECUTION MANAGEMENT

sdTEE execution is managed through a FIFO data structure, i.e., stack, where each invocation of a child sdTEE (sdTEE_C) involves pushing the sdTEE onto the execution stack using `push`. Upon completing the requested operations, the child sdTEE performs a `pop` operation to return execution to its parent (sdTEE_P). In case of an interrupt, depending on priority, the stack may need to be unwound to sdTEE_P. Resuming execution of sdTEE_C requires sdTEE_P to issue a call to AnyTEE, which results in the sdTEE_P handler performing a `push` operation.

2) FULL-FEATURED HYPERVISOR SUPPORT

AnyTEE operates in hypervisor mode, posing challenges in supporting software stacks featuring hypervisors. To overcome this, we introduce a hypervisor sdTEE (sdHyp) capable of hosting and creating guest VM sdTEEs (sdHyp_{VM}) as shown in Figure 4(A). Hypervisor support is achieved through nested virtualization techniques [39], [69], which can be implemented through para-virtualization, trap-and-emulation, or leveraging hardware support. Integration of sdHyp with other TEEs is possible and illustrated in Figure 4(B), for example, having sdTZ controlling the execution of sdHyp, and each sdHyp_{VM} hosting sdSGX enclaves. Integration with confidential computing TEEs, e.g., CoVE, is shown in Figure 4(C). CoVE features a Trusted Security Monitor (TSM) which creates and manages confidential VMs. The TSM is represented by sdTSM and controlled by sdHyp. sdHyp invokes the sdTSM to interact with confidential VMs (cVMs).

3) SDTEE-BASED SECURITY ENHANCEMENTS

By leveraging nested page tables, AnyTEE can leverage typical OS-level privileges and augment them to create flexible security primitives for applications. We propose three: 1) intra-privilege isolation, 2) read-only and write-once memory regions, and 3) immutable page tables. *1) Intra-privilege*

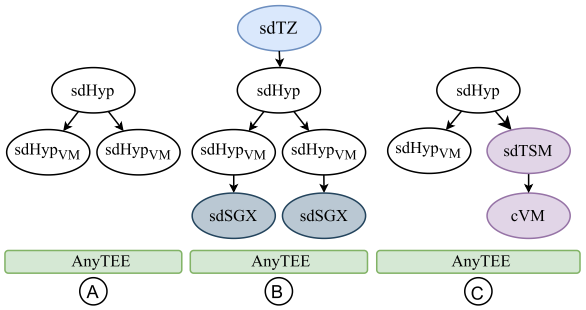


FIGURE 4. AnyTEE support for Hypervisor and TEE integration.

isolation establishes different permissions based on the current execution context, to restrict access to secrets based on what part of the application is executing. *2) Read-only and write-once memory regions* enables setting memory regions as read-only to the sdTEE, even if the first stage page table maps that memory as writable or disables virtual memory. The application must issue AnyTEE requests to mark pages as no longer writable. This mechanism can be used to maintain immutable log records. *3) Immutable page tables* renders the sdTEE page table memory region immutable and prevents the sdTEE from setting a new root page table. AnyTEE designates the page table memory region as read-only, inhibits modifications to the register holding the root page table address, and ensures that any writable memory region is not executable. This approach ensures no code injections are possible.

C. PROJECT CORE MECHANISMS

The AnyTEE core mechanisms consist of building blocks for TEE handlers, providing abstractions for sdTEE creation, management, attestation, and access to the host TEE on the platform. AnyTEE provides an event-based system to allow handlers to react to sdTEE events, including hypervisor and firmware calls, interrupts, and aborts. Table 2 summarizes the main interfaces the mechanisms expose.

1) SDTEE

sdTEEs can either be created at boot time or at run time. AnyTEE initialization routines will invoke `create` to create sdTEEs. `create` and `destroy` are used to dynamically create and destroy sdTEEs. `create` takes as input an sdTEE Manifest, while `destroy` takes an sdTEE UUID.

2) MEMORY MANAGEMENT

Spatial isolation is critical to prevent unauthorized access to TEE memory. TEEs have distinct requirements, ranging from fine-grain granularity (e.g., 4 kiB pages) to coarse-grain memory regions (e.g., 16 configurable memory regions for the entire system). Additionally, support for run-time modifications of the access control may also be necessary. To achieve this AnyTEE resorts to nested page tables, and provides `add_mem_region` and `remove_mem_region` functions. Memory management also supports exclusive IO device assignment to sdTEEs. To control direct memory accesses (DMA) by IO devices, memory management relies on platform protection mechanisms such as IOMMU or IOMPU. A challenge arises for sdTEEs with non-continuous physical address spaces (e.g., SGX, VM-based TEEs). In these cases AnyTEE leverages the IOMMU. In other cases, AnyTEE may be unable to support IO binding. To transfer resources between sdTEEs a handler must know the physical address of a resource, to obtain it, *Handlers* must use `mem_translate`.

3) EVENT MANAGER

AnyTEE provides a mechanism for handlers to subscribe to sdTEE events. Subscribing to firmware calls is possible using the `sub_firmware`, allowing handlers to react to requests with little or no modifications to existing software. Subscribing to hypervisor calls, `sub_hypercall`, allows handlers to establish their own hypercall interface. For example, to emulate SGX operations at the request of the GPOS. Furthermore, by subscribing accesses to specific memory regions, `sub_mem_region`, handlers can emulate IO devices. Events such as memory aborts can also be subscribed.

4) INTERRUPT MANAGEMENT

AnyTEE receives all system interrupts. A handler subscribes to its sdTEE interrupts through `sub_interrupts`. It also subscribes to interrupts of other sdTEEs, using the same function, e.g., to handle context switching between sdTEEs. Interrupts are then injected by its main handler using `inject_interrupt`.

5) ATTESTATION

To provide sdTEE attestation, we resort to an attestation evidence list that includes platform and sdTEE code UUIDs, software versions, resources attributed to the sdTEE, and a cryptographic hash of the sdTEE payload. Handlers leverage AnyTEE's attestation API, `get_quote`, to create attestation evidence. AnyTEE takes advantage of the host platforms' TEE or RoT to create hardware-bound public/private key pair, store the private key, and digitally sign the evidence list. The evidence list is then provided to a remote party, which will use it to decide whether to trust the sdTEE. Similar approaches have already been proposed [33].

6) HOST TEE INTERFACE

In some TEE models (e.g., TZ), primitives such as random number generation, secure storage, and cryptographic accelerators, may be hardwired to the platform's TEE. AnyTEE can provide these same features to sdTEEs through the Host TEE abstraction. For example, on a TZ-enabled platform, access to the replay-protected memory block (RPMB) needs to be mediated via a TA responsible for issuing secure storage requests. Alternatively, AnyTEE can use external RoT such as TPMs. To support unmodified binaries it may be necessary to emulate security-related IO devices that offer these features.

IV. IMPLEMENTATION

In this Section, we describe the implementation details to support (i) SGX and (ii) TZ TEE models, as well as (iii) sdTEE-based security enhancement. We implement sdSGX and sdTEE-based security enhancement on NXP's i.MX8MQ as Armv8-A target and sdTZ on both the i.MX8MQ and Digilent's Genesys 2 FPGA with a RISC-V CVA6 bitstream [54], [62]. We used the (Armv8-A) TF-A version 2.7, (RISC-V) OpenSBI 1.1, OPTEE version 3.21, and Linux 5.11.

A. SDSGX

In this section, we present our SGX TEE model (i.e., sdSGX) implementation for AnyTEE. Figure 5 depicts its architecture within AnyTEE. We also describe how sdSGX augments SGX security properties and extends its programming model.

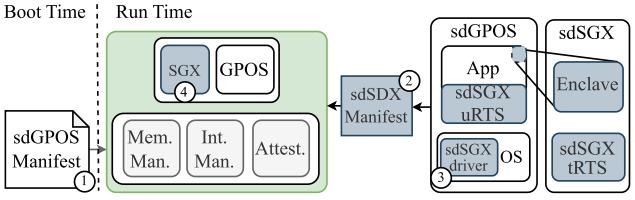


FIGURE 5. AnyTEE SGX implementation, sdSGX, including boot and run-time aspects, as well as sdSGX software.

1) SOFTWARE COMPONENTS

sdSGX implementation relies on a modified version of the SGX SDK and a custom SGX kernel driver. The untrusted runtime system (uRTS) was modified, creating sdSGX uRTS, to interact with our sdSGX kernel module. These modifications replace SGX functionality with calls to the SGX handler. We focus on the key functions: `sgx_create_enclave`, `sgx_destroy_enclave` and `sgx_ecall`, showcasing AnyTEE core features while not aiming for a complete SGX emulation. The kernel module is then responsible for interacting with the SGX handler. We built a unikernel runtime environment to support enclave execution in an sdTEE, based on the SGX SDK's trusted runtime system (tRTS), creating sdSGX tRTS. sdSGX tRTS interacts with the SGX handler, performs enclave

initialization, and manages invocation of enclave functions and calls to the untrusted application.

2) EXECUTION FLOW

Figure 5 illustrates the execution flow. AnyTEE loads an sdTEE Manifest to create an untrusted OS sdTEE (sdGPOS) during boot ①. The process of creating an sdSGX enclave involves: i) creating the sdTEE manifest, ii) loading it into memory, and iii) allocating the enclave memory regions ②. The application then requests AnyTEE SGX driver to create an enclave using the manifest ③. AnyTEE delivers requests to SGX handler ④, which authenticates the manifest, creates the sdTEE, and initializes it, setting the sdSGX as a child of sdGPOS. Ecalls and Ocalls allow applications to invoke enclave operations and vice-versa. In sdSGX, Ecalls are executed by requests to the sdSGX handler, through hypervisor calls, triggering context switches to the corresponding enclave. Ocalls require enclave access to the application's stack pointer to allocate Ocall arguments but otherwise operate similarly to Ecalls. In SGX, GPOS handles enclave exceptions and faults. In AnyTEE, the SGX handler handles interrupts and memory access faults, and signals such events to sdGPOS. The sdGPOS can then resume enclave execution through a resume call to the SGX handler.

3) AUGMENTED SGX

Side-channel protection (e.g., through cache and DRAM bank coloring) can be used to improve confidentiality. Although side-channel attacks are out of scope for AnyTEE, some of them can be minimized [63], [67] by removing sdGPOS' ability to map and unmap enclave memory regions at will and preventing granular enclave interrupts. sdSGX can improve availability by allowing prioritizing enclave execution or allocating custom time windows regardless of OS scheduling policies. We can also improve application confidentiality and integrity by preventing the enclave from having arbitrary access to the application's address space. Performance-wise, as sdSGX does not feature memory encryption, it improves native enclaves' performance. Additionally, there are no hard limits for memory region size available to sdSGX enclaves. sdSGX can also benefit from exclusive device assignment, reducing IO-related overhead compared to original SGX enclaves.

4) COMPATIBILITY AND LIMITATION

Our implementation adheres to the SGX programming model, allowing the execution of legacy enclaves without modifying architecture-independent enclave source code. However, the complete implementation tRTS, uRTS, enclave standard library code, and SGX Platform Software (PSW) has not been undertaken due to the effort required for support. Notably, multi-threaded enclaves are not supported in our current implementation. It's worth mentioning that sdSGX necessitates the pinning of application page tables to prevent swap operations. Given the lack of hardware encrypted memory, sdSGX also features a weaker threat model than SGX, e.g., an attacker with bus snooping capability could access the sdSGX's enclave state. However, AnyTEE could leverage memory encryption engines or SoC bound execution to achieve this property [35], [70], [71].

B. SDTZ

We now discuss AnyTEE implementation of sdTZ, illustrated in Figure 6, including support for OP-TEE trusted OS in RISC-V. Additionally, we explore enhancements to address some of TZ's architectural limitations [12].

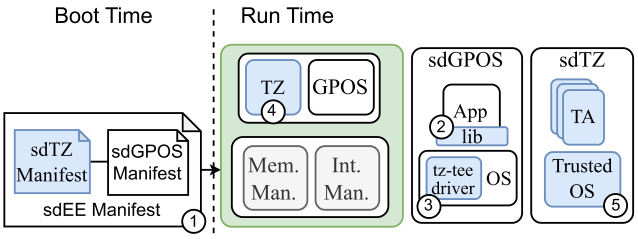


FIGURE 6. AnyTEE TZ implementation, sdTZ, including boot and run-time aspects, as well as sdTZ software.

1) SOFTWARE COMPONENTS

AnyTEE supports OP-TEE trusted OS on RISC-V, building upon existing porting efforts [1]. Some implementation details differ from the Arm architecture. For example, our OP-TEE RISC-V port doesn't support registering callbacks to different events, e.g., standard call (i.e., interruptible call) and fast call (i.e., uninterruptible call). Instead, in our implementation, the trusted OS distinguishes between events and invokes the appropriate handler. The TZ handler implements the context-switching logic to support communication between untrusted OS and OP-TEE. We modify OPTEE's Linux kernel driver to perform RISC-V Supervisor Binary Interface (SBI) calls instead of Arm Secure Monitor Calls (SMC). The AnyTEE handler then handles SBI calls. For exclusive access to MMIO devices, AnyTEE exclusively maps the MMIO region and binds associated interrupts to sdTZ.

2) EXECUTION FLOW

We instantiate two sdTEEs at boot time following TZ's boot flow, booting sdTZ software (secure world) before sdGPOS (normal world). The sdTEE manifest establishes resources belonging to each world ①. AnyTEE requires secure integration into the platform's secure boot. A request for TZ typically originates from an application ②. The application interacts with the trusted OS driver ③, which will then invoke the TZ handler ④, through SMC or SBI calls. The TZ handler then performs a context switch to the trusted OS ⑤. Our OP-TEE RISC-V port does not handle receiving interrupts meant for the GPOS. To address this, the *TZ handler* identifies such interrupts and forwards the event to the *GPOS handler*.

3) AUGMENTED TRUSTZONE

A critical security limitation of TZ is the trusted OS's access privileges [12], [13]. The trusted OS can access any normal and secure world resource. With AnyTEE it is possible to restrict the memory access privileges of trusted OSes running in sdTZs, preventing them from arbitrarily accessing resources of the sdGPOS sdTEE. Finally, AnyTEE allows the instantiation of multiple TZ-TEEs. Several works have targeted this topic [13], [29], [34], [37], with FFA on Armv8.4 also enabling this.

4) COMPATIBILITY AND LIMITATIONS

AnyTEE currently only supports OP-TEE trusted OS. Other trusted OSes may require direct support. We executed OPTEE's xtest (refer to §V-B), demonstrating wide compatibility.

C. INTRA-SDTEE ISOLATION

Recent CPU generations introduced mechanisms like Intel's Memory Protection Keys (MPK), nested page table switching with VM functions (VMFUNC), and Arm's Memory Tagging Extensions (MTE), for memory isolation [46]. In our project, we enhance sdSGX enclaves with rapid permission context-switching, akin to VMFUNC. This allows efficient management of access to sensitive information during execution addressing potential security risks in enclave implementations. Proposals to modify hardware for integrating isolation primitives into enclaves have already been explored [25].

The VMFUNC instruction on recent Intel processors enables VMs to modify the second stage page table root. Hypervisor-configured root pointers are stored in a table, allowing VM guests to select a specific root pointer. Invoking the VMFUNC instruction modifies the Guest Physical Addresses to Host Physical Addresses mappings by switching root pointers, allowing VMs to establish mappings that enable access to sensitive information only when needed, and efficiently grant and revoke access to it during execution. In AnyTEE, we implement an approach similar to VMFUNC, but that is implemented by software running at the OS level.

1) KEY INSIGHT

Our implementation starts from the observation that it is possible to efficiently switch address space, i.e., domains, with one write to root page table address register, TTBR0_EL1 in Armv8. An alternate approach would require updating page tables on every domain switch requiring page table modification and TLB invalidation. The key optimization to avoid page table modifications and TLB invalidation relies on pairing page tables and Address Space Identifiers (ASIDs), such that each ASID has a one-to-one relation with a page table (and vice-versa), and leveraging non-global page table property. Marking pages as non-global sets the page table as valid only for the respective ASID. Each page table and ASID pair form an execution context with different access permissions. This allows the TLB to select the correct page table entry for each domain. Not leveraging ASIDs and relying only on page table modifications can lead to incorrect memory access permissions. Cached page table entries in the TLB may result in using outdated entries, as the TLB deems them valid for the requested address translation, despite the page table switch.

2) ESTABLISHING DOMAINS

Each ASID represents a combination of permissions for each domain. For a setup with one domain where A is Accessible, W is Writable and * is Any, the following permissions are possible: $D_1\{AW\}$, $D_1\{A-\}$, $D_1\{-*\}$. To create this domain, three ASID and page table pairs are required. Supporting all combinations of N number of domains requires 3^N page table and ASID pairs. For 8-bit ASIDs, all combinations of 5 domains are supported, requiring 3^5 (243) page tables. For 16-bit ASIDs, all combinations of 10 domains require 3^{10} (59, 049) page tables. The number grows exponentially with N. It is up to the sdTEE developer to evaluate the trade-offs in speed in execution and number of page tables, considered in terms of memory utilization.

TABLE 3. Microbenchmarks for creation, destruction, and context switch operations, for an enclave with a binary size of 8.3 MiB.

	Creation	Destruction	Context Switch
Average	82.25 ms	212.40 ms	17.14 µs
Std. Dev.	0.83 ms	3.81 ms	1.71 µs
Coeff. of Var.	1.01%	1.79%	12.37%

V. EVALUATION

We evaluate sdSGX and sdTEE-based security enhancements on NXP's i.MX8MQ platform, as well as sdTZ on both the i.MX8MQ and Digilent's Genesys 2 FPGA with a RISC-V CVA6 bitstream, using microbenchmarks and real-world applications.[1]

A. SDSGX

For the **microbenchmarks**, we measure three key micro-operations: i) enclave creation, ii) enclave destruction, and iii) context-switching. The evaluated enclave was configured with a memory region of 8.3 MiB (enclave binary size). For the remaining tests, we compare Linux and sdSGX versions of the same application. We evaluate AnyTEE's sdSGX impact on CPU-intensive workloads by running nbench [25], [33]. Next, we measure the execution time of read operations of an application implementing an SSL server. Client requests originate from co-located client applications running on the same OS instance. At the **application level** we create an application that receives 300 kiB of encrypted data in 16 kiB chunks and for every chunk sends an acknowledgment message.

[1] We validated the execution of simultaneous TEEs. However, for the sake of simplicity, we evaluate sdTZ and sdSGX independently.

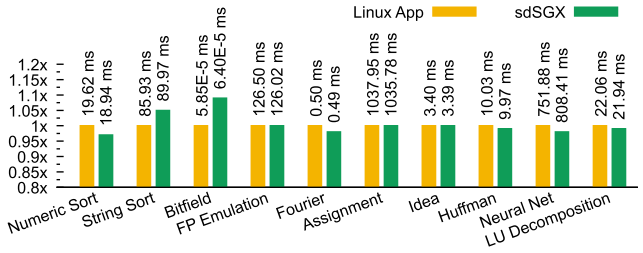


FIGURE 7. Relative and absolute execution time for nbench benchmark workloads for Linux App and sdSGX.

1) METHODOLOGY

To measure creation, destruction, and context switch operations, we use Linux time API and register elapsed time for the execution of each test (CLOCK_REALTIME parameter). For nbench, we look only at the reported score values. To measure the execution of SSL operations we use the architectural timer to avoid context switches between the enclave and application. For nbench and SSL, we compare execution between Linux and sdSGX versions of the same applications.

2) MICROBENCHMARKS

Table 3 summarizes the results for the micro-operations. Enclave creation requires nested page table modifications and, thus, TLB invalidation. In enclave destruction, the entire enclave memory is zeroed to prevent leaks of security-sensitive data, and second-stage page tables are modified. Finally, the context-switch involves context switches from the application to the OS, from the OS to AnyTEE, and from AnyTEE to sdSGX.

SGX-nbench results show negligible overhead, Figure 7, with the geometric mean of the relative execution time for the SGX-nbench being approximately 0.01× when compared to the nbench for Linux. These results align with recent reports on the performance of hardware virtualization [42]. Lastly, for the SSL read operation, Figure 8, the results indicate that the context-switch operation has a non-negligible impact, 1.39×, especially for smaller buffers. With smaller buffers, the fixed cost of each context switch represents a larger proportion of the total execution time, resulting in a higher relative overhead.

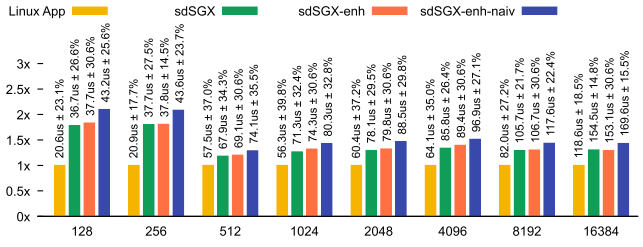


FIGURE 8. Execution of OpenSSL read operations for different buffer sizes. The four scenarios are: Linux application (native), sdSGX enclave (sdSGX), sdSGX enclave optimized enhancement (sdSGX-enh), sdSGX enclave with naive enhancement (sdSGX-enh-naiv).

TABLE 4. The OpenSSL operation reads 300KB of data in 16KB chunks. The scenarios are: Linux (native), sdSGX enclave (sdSGX), sdSGX enclave with optimized enhancement (sdSGX-enh), and sdSGX enclave with naive enhancement (sdSGX-enh-naiv).

	Overhead	Avg. (ms)	Coeff. of Var.	Ctx. Switches
Linux App	-	25.80	2.30%	–
sdSGX	3.16%	26.61	2.20%	81
sdSGX-enh	3.20%	26.64	2.90%	81
sdSGX-enh-naiv	13.70%	29.34	2.40%	81

3) APPLICATION

Table 4 presents results for the SGX-OpenSSL enclave application. Overall the performance is near-native, introducing on average less than 3% overhead. This is justified by the residual impact the context switch and hardware virtualization have on overall execution time of the enclave.

B. SDTZ

Due to potential deviations between the performance overhead on an FPGA-based soft-core[2] and a hard-core in silicon, we have complemented the results with experiments conducted on the COTS Arm NXP platform, which is endowed with a sophisticated and performing microarchitecture. For **microbenchmarks**, we measure world switch time and the execution time of OP-TEE xtest 4.0.0, a suite of OP-TEE regression tests (grouped into nine categories) to validate the implementation of the trusted kernel and APIs. xtest also includes benchmarks to assess the performance of cryptographic and storage-related operations. At the macro level, we measure the execution time of a **real-world application**, an open-source Bitcoin wallet TA [31].

1) METHODOLOGY

We use the architectural timer to measure context switch time. For xtest and Bitcoin TA, we used Linux time API with CLOCK_REALTIME parameter. We compare sdTZ to two TEE implementations, a typical TZ scenario on Arm leveraging TF-A, and our own support for TEE on RISC-V with a modified OpenSBI firmware.

2) MICROBENCHMARKS

World switch time with sdTZ (Table 5) is reduced compared to OpenSBI by 3.5%. This improvement is mostly due to the need to perform TLB invalidation on context switches using OpenSBI. On average, the performance overhead was 5%, with a maximum of up to 13% (Figure 9). We hypothesized this is mostly due to CVA6's simple microarchitecture, and corroborated this hypothesis by running sdTZ on Arm (with hardware virtualization support). For the AnyTEE Aarch64, we observed only a performance overhead of around 1%; this is in line with other recent studies [42], and the expectation for RISC-V silicon when hardware virtualization extensions

[2]Previous work has exposed that hardware virtualization support on CVA6 is not optimized for performance due to the simple microarchitecture [54].

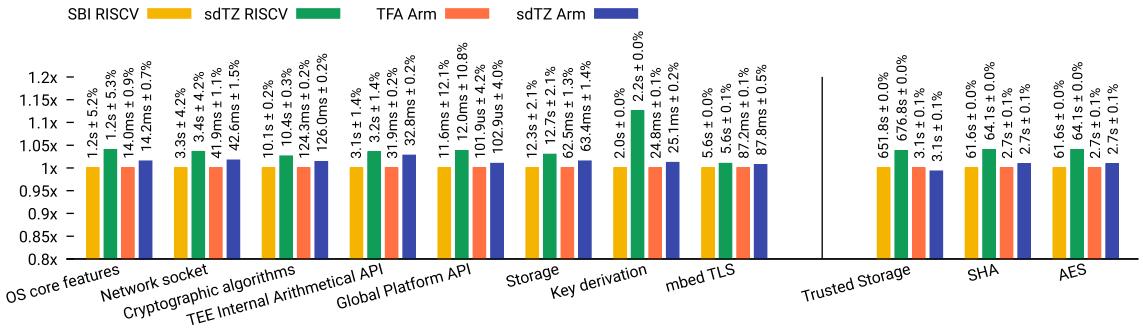


FIGURE 9. Geometric mean of sdTZ overhead for xtest. Each bar represents the geometric mean of overheads for the tests within a group. The left side of the figure presents the unit tests for the nine groups. The right side of the figure presents results for the benchmarks in three groups.

become available in COTS platforms (e.g., SiFive P800 Series, Microchip PolarFire 2).

TABLE 5. Context switch for transition from GPOS to trusted OS.

	Average	Std. Dev.	Coeff. of Var.
sdTZ RISCV	209.03 µs	17.26 µs	8.26%
SBI RISCV	216.74 µs	20.18 µs	9.31%
Relative	0.96x	-	-

3) APPLICATION

Figure 10 presents the results for the Bitcoin wallet TA. The geometric mean of the performance overhead is 1.16% for RISC-V and 0.2% for Aarch64, lower than the overhead assessed with xtest. This is explained by the fact that the operations performed by the Bitcoin wallet TA are not small, isolated tasks. Instead, they include broader application logic, which reduces the relative impact of context switching overhead over the application's runtime. As a result, we conclude that AnyTEE has negligible impact in real-world applications.

C. INTRA-SDTEE ISOLATION

We evaluate the performance of sdSGX with additional intra-sdTEE isolation (Figure 8 and Table 4). We evaluate an optimized implementation, i.e., page table switches are performed using a single instruction, denoted as sdSGX-enh. The naive implementation, involving modifications to page table entries and TLB invalidation operations, is sdSGX-enh-naiv.

1) MICROBENCHMARK

Figure 8 shows the overheads for the two implemented intra-sdTEE isolation approaches. The results show that the optimized version performs 9.14% better than the naive approach, explained by the cost of performing TLB invalidations. Overall, sdSGX-enh incurs 1.87% overhead compared to the plain sdSGX enclave.

2) APPLICATION

In Table 4, the overheads of both implemented intra-sdTEE isolation approaches are presented for receiving a large buffer. Results report that the optimized version outperforms the naive approach by 9.2%. Again, this speedup can be justified by the reduced number of expensive microarchitectural operations, i.e., TLB invalidation and subsequent decrease in TLB misses. Overall, compared to the vanilla sdSGX enclave, sdSGX-enh adds 0.08% overhead.

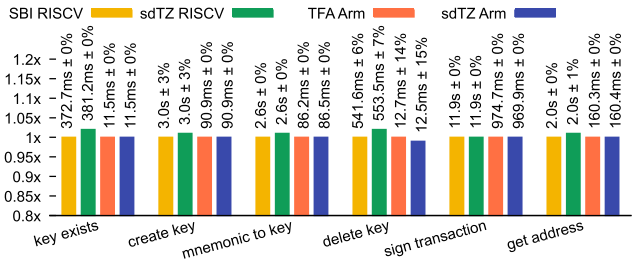


FIGURE 10. Relative, absolute execution time, and normalized standard deviation for Bitcoin wallet commands.

VI. SECURITY EVALUATION

We evaluate AnyTEE security through a theoretical analysis of its enforcement of security properties. We address the vectors that may be attempted to subvert security properties and highlight the protective mechanisms enforced by AnyTEE.

A. A1. COMPROMISED GPOS

In case the GPOS is compromised, AnyTEE's nested page tables and IO protection mechanisms prevent it from making arbitrary accesses. A GPOS can abuse enclave loading. To prevent this the sdSGX handler confirms resource ownership, and attests the enclave binary. The GPOS can try to scan the enclave's secrets after destruction. These attempts are thwarted by zeroing the enclave's memory and invalidating the cache upon enclave destruction.

B. A2. COMPROMISED ENCLAVE

A compromised enclave may target the host application within its threat model. However, attempts to compromise the

GPOS, AnyTEE's hypervisor, or other sdTEE components are prevented due to nested page tables and 1O protection mechanisms.

C. A3. COMPROMISED TRUSTED OS

A compromised trusted OS may try to gain control over the GPOS, as allowed in TZ's threat model. However, with proper configuration, AnyTEE allows restricting trusted OS access, and creating multiple sdTEEs for additional trusted OS environments. Recent works have addressed the importance of sandboxing trusted OSes to mitigate several critical CVEs [13], [37].

D. A4. TCB TAMPERING

AnyTEE safeguards against TCB tampering by employing secure boot to abort the system boot sequence in case of firmware integrity check failure. Additionally, AnyTEE relies on one-time programmable memories, such as eFuses, to prevent attackers from modifying the platform's certificates and compromising the chain of trust.

E. A5. PHYSICAL ACCESS

AnyTEE does not protect against physical access attacks. However, similarly to other works, AnyTEE can use software-based solutions to encrypt memory assuming the presence of on-chip memory [35], [70], or use memory encryption accelerators if present.

F. A6. SIDE CHANNEL

While side-channel attacks are outside the scope of AnyTEE, mitigation techniques (e.g., cache coloring) can be used at the expense of some performance degradation. Controlled channel attacks can be mitigated by adopting more secure behavior for TEE emulation, for example, removing the ability of GPOS to unmap enclave memory [67] or to interrupt enclave with high-precision interrupts [63].

VII. DISCUSSION

A. CONFIDENTIAL COMPUTING MODEL SUPPORT

Confidential computing solutions leverage an untrusted hypervisor that manages the platform's resources and allocates memory to create confidential VMs. Although we currently don't implement support for this, the core mechanisms provide a foundation that enables its implementation. In SEV the untrusted hypervisor allocates and manages confidential VMs. In AnyTEE, this can be implemented by extending the nested-virtualization handler with the ability to create VMs that are managed but inaccessible to the untrusted hypervisor. In TDX, CCA, and CoVE, an untrusted hypervisor interacts with a second hypervisor, referred to as a monitor. In this case, a new AnyTEE handler dedicated to emulating the interaction with the confidential VM monitor must be added.

B. PER-CORE SOFTWARE STACKS ASSIGNMENT

While not the primary objective of AnyTEE, its design inherently allows for the assignment of software stacks on a per-core or set of cores basis, each equipped with its individual sdTEEs, thereby enabling distinct security properties for each stack. Compared to the simultaneous execution of independent software stacks on the same cores, micro-architectural isolation is simplified by reducing shared functional units. Architectural isolation is also streamlined, eliminating the need for complex scheduling mechanisms. Minimizing the sharing of hardware devices between software stacks further simplifies implementation. Another advantageous use case is the allocation of dedicated cores for security functionality. The incorporation of heterogeneous software stacks enhances system security by facilitating fine-grained decomposition of system functionality.

VIII. RELATED WORK

A. ANYTEE IN PERSPECTIVE

We compare AnyTEE with works that aim to provide TEE compatibility or emulation and configurable TEEs. This comparison is presented in Table 6. We categorize these systems by evaluating the features they introduce. If a feature falls outside the work's defined scope or is left unaddressed, we interpret the system as performing negatively in that aspect.

1) TEE MODEL AND COMPATIBILITY

TEE emulation has mainly focused on single TEE compatibility. For SGX, efforts like HyperEnclave [33], vSGX [72], and Komodo [20] extend enclave support beyond Intel platforms. CCA and confidential VM emulation enable confidential computing on Arm TZ platforms without realm extensions, as proposed by VirtCCA [66] and TwinVisor [36]. vTZ [29] and TEEVsel4 [10] emulate TZ for per VM secure environments, while MyTEE [28] addresses TZ controller absence in low-cost Arm platforms. AnyTEE can simultaneously emulate multiple TEEs, hosting unmodified TEE software binaries, if running on the same architecture, provided that the handler emulates the TEE behavior and the necessary devices. Keystone does not support existing TEE software, instead it uses custom runtimes for enclave applications, accommodating unmodified GPOS applications, Keystone enclave applications, and isolated OSes. Cure supports various TEE models but without aiming to support existing TEE software stacks. AnyTEE provides similar levels of TEE models, and uses them to support multiple existing TEEs. Although support for all mentioned TEE models is not implemented, AnyTEE lays a foundation that enables future support. AnyTEE also allows nesting TEEs such that each TEE can delegate execution to its own TEEs offering new degrees of configuration and flexibility. Additionally, AnyTEE's flexibility allows modeling future TEE models.

TABLE 6. TEE emulation and configurability state of the art comparison. Key: Phys. - Physical; SC - Side-Channel; DoS - Denial-of-Service; CC - Controlled Channel; ● - fares positively; ◐ - fares partially positively, ✗ - fares negatively; — - does not apply; f.v. - formally verified; n.p. - not provided.

		TEE Model				TEE Compatibility								HW Dependency		Attack				TCB
		User	Kernel	Intra	Nested	TZ	FFA	CCA	SGX	TDX	SEV	RISC-V	CoVE	COTS	Cross.	Phys.	SC	DoS	CC	kLoC
TEE Emulation	vTZ [29]	○	●	○	○	●	○	○	○	○	○	○	○	●	○	○	○	●	○	2
	TEEVseL4 [10]	○	●	○	○	●	○	○	○	○	○	○	○	●	◐	○	○	○	○	n.p.
	MyTEE [28]	○	●	○	○	●	○	○	○	○	○	○	○	●	◐	○	○	○	○	5.5
	TwinVisor [36]	○	●	○	○	○	○	◐	○	○	○	○	○	●	○	○	○	○	○	8.7
	virtCCA [66]	○	●	○	○	○	○	●	○	○	○	○	○	●	○	○	○	●	○	11.9
	vSGX [72]	●	○	○	○	○	○	○	●	○	○	○	○	●	○	●	○	○	○	16.3
	Komodo [20]	●	○	○	○	○	○	○	◐*	○	○	○	○	●	○	○	○	○	●	3.6 f.v.
	HyperEnclave [33]	●	◐	○	○	○	○	○	●	○	○	○	○	●	◐	◐	◐	○	●	7.5
Configurable TEE	Keystone [35]	●	●	○	○	—	—	—	—	—	—	—	—	●	○	◐	◐	◐	●	10.7
	Cure [9]	●	●	●	○	—	—	—	—	—	—	—	—	○	○	○	◐	◐	●	3.1
Configurable TEE & Emulation	**AnyTEE**	●	●	●	●	●	◐	◐	●	◐	◐	◐	◐	●	●	◐	◐	●	●	9.4+

2) HW DEPENDENCIES

Although most systems leverage COTS platforms, they often cannot be deployed across ISAs, as they leverage specific TEEs such as TZ [20], [29], [36], [66] and AMD SEV [72]. Hyperenclave targets x86 platforms and can benefit from AMD SME or Intel MKTME to offer memory encryption; however, support for Arm and RISC-V ISAs is not implemented. MyTEE leverages common virtualization and debug features, but requires significant engineering effort on IO access interposition for DMA-capable devices. Current configurable TEE solutions are not scalable, i.e., Keystone leverages RISC-V ISA's PMP restricting the solution to RISC-V platforms and to the limitations of the mechanism, mainly the limited number of regions. Cure on the other hand proposes specialized hardware to create TEEs. AnyTEE capitalizes on the widespread hardware support for DMA device isolation and virtualization in common ISAs.

3) ATTACKS

Regarding attacks, most works consider side-channel mitigation out of scope. Preventing denial-of-service requires that the system can take control of execution, and multiple works can implement this, but for the majority it falls outside of the scope. To prevent physical attacks, e.g., bus snooping, cold boot, systems must use memory encryption. Although Keystone offers protection for L1 cache side-channels it cannot provide L2 cache isolation without specific hardware support. Cure provides mechanisms for L1 and L2 cache isolation. AnyTEE does not implement L1 or L2 cache isolation, but it can be adapted to support isolation schemes. One example is using cache coloring and reserving one cache color for TEEs, with all other colors for untrusted sdTEEs, and forcing only one TEE to execute at a time.

4) TCB

Number of lines of code (LoC) ranges from 2 kLoC to 16.3 kLoC, including various layers from firmware to hypervisor. For TEEVseL4, seL4 is formally verified but the VMM, for which LoC is not provided, is not. Keystone TCB encompasses the secure monitor software and in Cure only a subset of the secure monitor code. The core of AnyTEE is 9.4 kLoC, with additional TEE emulation handlers requiring additional lines of code. For sdTZ, we add 131 LoC and for sdSGX we add 413 LoC. AnyTEE also trusts the host's TEE.

B. OTHER WORKS

Some systems offer specific hardware and firmware components for constructing TEE systems [56]. These include CPU extensions such as TZ, SGX, SEV, and others [2], [9], [18], [30], [47], [51], [53], dedicated co-processors [9], [41], [52], tiles of security processors [5], [50], [65], and security chips such as Titan-M and TPM [23], [45]. Academia has proposed to leverage the available TEE primitives to achieve improved security goals, creating isolated environments with improved security guarantees, e.g., Sanctuary, ReZone, and others [11], [13], [20], [26], [59]. Other works leverage virtualization in addition to the platform's security mechanisms [15], [26], [32], [73], with some, like TEEv [37] and PrOS [34] leveraging software-based virtualization techniques. Overshadow [14] and Inktag [44], among others [43], [49], [68] have proposed the use of virtualization as the single isolation mechanism. Use of formal techniques to create TEEs, e.g., [3], [22], addresses low-end devices. To improve on interoperability, initiatives such as Google's Asylo [7], Enarx [19] and others [6], [16], [24], [57], [58], [60], [61], provide specific SDKs. However, re-writing code to target an SDK, and relying on the SDK to support the selected platform precludes using legacy TEE applications with minimal effort. Recent work has also applied TEE-related trust and privacy principles in domain-specific settings. For instance, TFL-DT [27] proposes a trust evaluation scheme for federated learning. Similarly, PPRU [40] introduces a privacy-preserving reputation update mechanism for cloud-assisted vehicular networks. These efforts extend TEE concepts into real-world collaborative systems, emphasizing trust and privacy in distributed environments.

IX. CONCLUSION

In this work, we propose AnyTEE, a framework for developing software defined TEEs that builds upon a set of virtualization-based security mechanisms. We achieve cross-platform, simultaneous, and refined software defined TEEs,

compatible with legacy TAs. Our evaluation demonstrates overheads of less than 3% when compared to native TEE solutions while achieving cross-platform compatibility with two of the widely used TEEs. While AnyTEE delivers low overhead and supports a wide range of platforms, adding support for additional TEE architectures (e.g., RISC-V CoVE) would improve its applicability. Furthermore, tighter integration with platform security features such as secure boot and attestation primitives, along with formal verification of the virtualization layer, would strengthen its security guarantees.

REFERENCES

[1] (5734). *Riscv: Initial Support of Qemu Risc-v Generic Virtual Platform (virt)*. [Online]. Available: https://github.com/OP-TEE/optee_os/pull/5734

[2] AMD. (2019). *AMD Secure Encrypted Virtualization*. [Online]. Available: www.developer.amd.com/sev/

[3] M. Ammar, B. Crispo, B. Jacobs, D. Hughes, and W. Daniels, "S the security MicroVisor: A formally-verified software-based security architecture for the Internet of Things," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 5, pp. 885–901, Sep. 2019.

[4] *Arm Firmware Framework for Arm A-profile*, Arm, Cambridge, U.K., 2024. [Online]. Available: https://developer.arm.com/documentation/den0077/latest

[5] M. Armanuzzaman, A.-R. Sadeghi, and Z. Zhao, "Building your own trusted execution environments using FPGA," 2022, *arXiv:2203.04214*.

[6] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux containers with Intel SGX," in *Proc. USENIX OSDI*, Nov. 2016, pp. 689–703.

[7] (2022). *Asylo*. [Online]. Available: www.github.com/google/asylo

[8] (2024). *AnyTEE Authors*. [Online]. Available: https://github.com/anyteeauthors/anytee

[9] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A. Sadeghi, and E. Stapf, "CURE: A security architecture with CUstomizable and resilient enclaves," in *Proc. USENIX Secur.*, Jan. 2020, pp. 1–12.

[10] B. Blazevic, M. Peter, M. Hamad, and S. Steinhorst, "TEEVseL4: Trusted execution environment for virtualized seL4-based systems," in *Proc. IEEE 29th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2023, pp. 67–76.

[11] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with user-space enclaves," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–21.

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

[13] D. Cerdeira, J. Martins, N. C. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE privilege reduction," in *Proc. USENIX Secur.*, Jan. 2022, pp. 2261–2279.

[14] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 2, pp. 2–13, Mar. 2008.

[15] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *Proc. USENIX ATC*, Jun. 2016, pp. 565–578.

[16] Confidential Computing Consorcium. (2022). *Veracruz: Privacy-preserving Collaborative Compute*. [Online]. Available: www.github.com/veracruz-project/veracruz

[17] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, Jan. 2016.

[18] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. USENIX Secur.*, Jan. 2016, pp. 857–874.

[19] (2022). *Enarx*. [Online]. Available: www.github.com/enarx/enarx

[20] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 287–305.

[21] RISC-V Found. (2021). *Privileged Architecture V1.12, Ratified*. [Online]. Available: www.github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12

[22] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo, "PISTIS: Trusted computing architecture for low-end embedded systems," in *Proc. USENIX Secur.*, 2022, pp. 3843–3860.

[23] Trusted Computing Group. (2018). *TPM*. [Online]. Available: www.trustedcomputinggroup.org/work-groups/trusted-platform-module/

[24] J.-Y. Gu, H. Li, Y.-B. Xia, H.-B. Chen, C.-G. Qin, and Z.-Y. He, "Unified enclave abstraction and secure enclave migration on heterogeneous security architectures," *J. Comput. Sci. Technol.*, vol. 37, no. 2, pp. 468–486, Apr. 2022.

[25] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, "A Hardware–Software co-design for efficient intra-enclave isolation," in *Proc. USENIX Secur.*, 2022, pp. 3129–3145.

[26] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure execution of unmodified applications with ARM TrustZone," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2017, pp. 488–501.

[27] J. Guo, Z. Liu, S. Tian, F. Huang, J. Li, X. Li, K. K. Igorevich, and J. Ma, "TFL-DT: A trust evaluation scheme for federated learning in digital twin for mobile networks," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 11, pp. 3548–3560, Nov. 2023.

[28] S.-K. Han and J. Jang, "MyTEE: Own the trusted execution environment on embedded devices," in *Proc. NDSS*, 2023, pp. 1–15.

[29] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "VTZ: Virtualizing ARM TrustZone," in *Proc. USENIX Secur.*, Jan. 2017, pp. 541–556.

[30] Intel. (2021). *Intel TDX*. [Online]. Available: intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html

[31] J. Cui. (2018). *Bitcoin Wallet Implementation Using Trusted Execution Environments*. [Online]. Available: www.github.com/Jamie-Cui/bitcoin-wallet

[32] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "PrivateZone: Providing a private execution environment using ARM TrustZone," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 5, pp. 797–810, Sep. 2018.

[33] Y. Jia, S. Liu, W. Wang, Y. Chen, Z. Zhai, S. Yan, and Z. He, "HyperEnclave: An open and cross-platform trusted execution environment," in *Proc. USENIX ATC*, Jan. 2022, pp. 437–454.

[34] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "PrOS: Light-weight privatized se cure OSes in ARM TrustZone," *IEEE Trans. Mobile Comput.*, vol. 19, no. 6, pp. 1434–1447, Jun. 2020.

[35] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16.

[36] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan, "TwinVisor: Hardware-isolated confidential virtual machines for ARM," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, Oct. 2021, pp. 638–654.

[37] W. Li, Y. Xia, L. Lu, H. Chen, and B. Zang, "TEEv: Virtualizing trusted execution environments on mobile platforms," in *Proc. 15th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, Apr. 2019, pp. 2–16.

[38] X. Li, X. Li, C. Dall, R. Gu, J. Nieh, Y. Sait, and G. Stockwell, "Design and verification of the arm confidential compute architecture," in *Proc. USENIX OSDI*, 2022, pp. 465–484.

[39] J. T. Lim, C. Dall, S.-W. Li, J. Nieh, and M. Zyngier, "NEVE: Nested virtualization extensions for ARM," in *Proc. 26th Symp. Operating Syst. Princ.*, Oct. 2017, pp. 201–217.

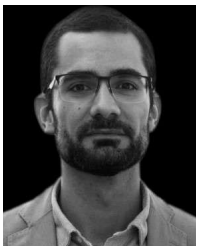
[40] Z. Liu, L. Wan, J. Guo, F. Huang, X. Feng, L. Wang, and J. Ma, "PPRU: A privacy-preserving reputation updating scheme for cloud-assisted vehicular networks," *IEEE Trans. Veh. Technol.*, vol. 74, no. 2, pp. 1877–1892, Feb. 2025.

[41] T. Mandt, M. Solnik, and D. Wang, "Demystifying the secure enclave processor," Black Hat, Las Vegas, NV, USA, Tech. Rep., 2016. [Online]. Available: https://www.blackhat.com/us-16/briefings.html#demystifying-the-secure-enclave-processor

[42] J. Martins and S. Pinto, "Shedding light on static partitioning hypervisors for arm-based mixed-criticality systems," in *Proc. IEEE 29th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, May 2023, pp. 40–53.

[43] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 143–158.

[44] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: Secure applications on an untrusted operating system," in *Proc. 18th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2013, pp. 265–278.
[45] D. Melotti, M. Rossi-Bellom, and A. Continella, "Reversing and fuzzing the Google Titan M chip," in *Proc. Reversing Offensive-Oriented Trends Symp.*, Nov. 2021, pp. 1–10.
[46] V. Narayanan and A. Burtsev, "The opportunities and limitations of extended page table switching for fine-grained isolation," *IEEE Secur. Privacy*, vol. 21, no. 3, pp. 16–26, May 2023.
[47] P. Nasahl, R. Schilling, M. Werner, and S. Mangard, "HECTOR-V: A heterogeneous CPU architecture for a secure RISC-V execution environment," in *Proc. Asia CCS*, May 2021, pp. 187–199.
[48] (2024). *OpenEuler*. [Online]. Available: https://docs.openeuler.org/en/docs/22.03_LTS/docs/secGear/introduction-to-secGear.html
[49] S. Pereira, J. Sousa, S. Pinto, J. Martins, and D. Cerdeira, "Bao-enclave: virtualization-based enclaves for arm," in *Proc. IEEE 8th World Forum Internet Things (WF-IoT)*, Oct. 2022, pp. 1–6.
[50] S. Pereira, D. Cerdeira, C. Rodrigues, and S. Pinto, "Towards a trusted execution environment via reconfigurable FPGA," 2021, *arXiv:2107.03781*.
[51] S. Pinto and N. Santos, "Demystifying Arm TrustZone: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 1–36, 2019.
[52] Qualcomm. (2019). *Qualcomm Secure Processing Unit SPU230*. [Online]. Available: www.216.117.4.138/files/epfiles/1045b_pdf.pdf
[53] Arm. (2021). *Realm Management Extension*. [Online]. Available: www.developer.arm.com/documentation/den0126/latest
[54] B. Sá, L. Valente, J. Martins, D. Rossi, L. Benini, and S. Pinto, "CVA6 RISC-V virtualization: Architecture, microarchitecture, and design space exploration," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 31, no. 11, pp. 1713–1726, Nov. 2023.
[55] R. Sahita, V. Shanbhogue, A. Bresticker, A. Khare, A. Patra, S. Ortiz, D. Reid, and R. Kanwal, "CoVE: Towards confidential computing on RISC-V platforms," in *Proc. 20th ACM Int. Conf. Comput. Frontiers*, May 2023, pp. 315–321.
[56] M. Schneider, R. Jayaram Masti, S. Shinde, S. Capkun, and R. Perez, "SoK: Hardware-supported trusted execution environments," 2022, *arXiv:2205.12742*.
[57] (2022). *Open Enclave SDK*. [Online]. Available: www.github.com/openenclave/openenclave
[58] (2022). *SOFAEnclave*. [Online]. Available: www.github.com/SOFAEnclave/SOFAEnclave
[59] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "TrustICE: Hardware-assisted isolated computing environments on mobile devices," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2015, pp. 367–378.
[60] Teaclave. (2022). *Teaclave: A Universal Secure Computing Platform*. [Online]. Available: www.github.com/apache/incubator-teaclave
[61] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proc. USENIX ATC*, Jul. 2017, pp. 645–658.
[62] L. Valente, A. Veeran, M. Sinigaglia, Y. Tortorella, A. Nadalini, N. Wistoff, B. Sá, A. Garofalo, R. Psiakis, M. Tolba, A. Kulmala, N. Limaye, O. Sinanoglu, S. Pinto, D. Palossi, L. Benini, B. Mohammad, and D. Rossi, "Shaheen: An open, secure, and scalable RV64 SoC for autonomous nano-UAVs," in *Proc. Hot Chips Symp. (HCS)*, 2023, pp. 1–12.
[63] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-step: A practical attack framework for precise enclave execution control," in *Proc. 2nd Workshop Syst. Softw. Trusted Execution*, Oct. 2017, pp. 1–6.
[64] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an eXtensible and modular hypervisor framework," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 430–444.
[65] C. Weinhold, N. Asmussen, D. Göhringer, and M. Roitzsch, "Towards modular trusted execution environments," in *Proc. 6th Workshop Syst. Softw. Trusted Execution*, May 2023, pp. 10–16.
[66] X. Xu, W. Wang, Y. Wu, C. Wang, H. Zhu, H. Ma, Z. Min, Z. Pang, R. Hou, and Y. Jin, "VirtCCA: Virtualized arm confidential compute architecture with TrustZone," 2023, *arXiv:2306.11011*.
[67] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 640–656.
[68] J. Yang, "Using hypervisor to provide application data secrecy on a per-page basis," in *Proc. Virtual Execution Environments (VEE)*, 2008.
[69] F. Zhang, J. Chen, H. Chen, and B. Zang, "CloudVisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proc. 33rd ACM Symp. Operating Syst. Princ.*, Oct. 2011, pp. 203–216.
[70] M. Zhang, Q. Zhang, S. Zhao, Z. Shi, and Y. Guan, "SoftME: A software-based memory protection approach for TEE system to resist physical attacks," *Secur. Commun. Netw.*, vol. 2019, pp. 1–12, Mar. 2019.
[71] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "SecTEE: A software-based approach to secure enclave architecture using TEE," in *Proc. CSS*, 2019, pp. 1723–1740.
[72] S. Zhao, M. Li, Y. Zhangyz, and Z. Lin, "VSGX: Virtualizing SGX enclaves on AMD SEV," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 321–336.
[73] 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.



DAVID CERDEIRA received the Ph.D. degree in electronics and computer engineering. He is currently a Researcher with the Embedded Systems Research Group, University of Minho, Portugal. His research interests include trusted execution environments, embedded systems, embedded security, and virtualization. He has contributed to these fields through several peer-reviewed publications.



JOSÉ MARTINS received the master's degree in electronics and computer engineering. During the master's degree, he was a Visiting Student with the University of Wurzburg, Germany. He is currently pursuing the Ph.D. degree with the Embedded Systems Research Group, University of Minho, Portugal. He has a significant background in operating systems and virtualization for embedded systems. He is the main author of the Bao hypervisor.



NUNO SANTOS (Member, IEEE) received the Ph.D. degree from the Max Planck Institute for Software Systems (MPI-SWS), Saarland University. He is currently an Associate Professor with the Department of Computer Science and Engineering (DEI), Instituto Superior Técnico (IST), University of Lisbon, and a Senior Researcher with INESC-ID Lisbon. He is a member of the Distributed, Parallel, and Secure Systems (DPSS) Group. His research has been published in top-tier conferences.



SANDRO PINTO received the Ph.D. degree in electronics and computer engineering. He is currently an Associate Research Professor with the ALGORITMI Research Centre, Universidade do Minho, Portugal. He has a deep academic background and several years of industry collaboration focusing on virtualization, security for embedded systems, and the IoT systems. He has published 100 peer-reviewed articles and is a skilled presenter with speaking experience in top-tier academic and industrial conferences.