CURE: A Security Architecture with CUstomizable and Resilient Enclaves

Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf,
Technische Universität Darmstadt

https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

CURE: A Security Architecture with CUstomizable and Resilient Enclaves

Raad Bahmani, Ferdinand Brasser, Ghada Dessouky,
Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, Emmanuel Stapf
Technische Universität Darmstadt, Germany
{raad.bahmani, ferdinand.brasser, ghada.dessouky, patrick.jauernig,}
{matthias.klimmek, ahmad.sadeghi, emmanuel.stapf}@trust.tu-darmstadt.de

Abstract

Security architectures providing Trusted Execution Environments (TEEs) have been an appealing research subject for a wide range of computer systems, from low-end embedded devices to powerful cloud servers. The goal of these architectures is to protect sensitive services in isolated execution contexts, called *enclaves*. Unfortunately, existing TEE solutions suffer from significant design shortcomings. First, they follow a *one-size-fits-all* approach offering only a single enclave *type*, however, different services need flexible enclaves that can adjust to their demands. Second, they cannot efficiently support emerging applications (e.g., Machine Learning as a Service), which require secure channels to peripherals (e.g., accelerators), or the computational power of multiple cores. Third, their protection against cache side-channel attacks is either an afterthought or impractical, i.e., no fine-grained mapping between cache resources and individual enclaves is provided.

In this work, we propose CURE, the first security architecture, which tackles these design challenges by providing different types of enclaves: (i) *sub-space* enclaves provide vertical isolation at all execution privilege levels, (ii) *user-space* enclaves provide isolated execution to unprivileged applications, and (iii) *self-contained* enclaves allow isolated execution environments that span multiple privilege levels. Moreover, CURE enables the exclusive assignment of system resources, e.g., peripherals, CPU cores, or cache resources to single enclaves. CURE requires minimal hardware changes while significantly improving the state of the art of hardware-assisted security architectures. We implemented CURE on a RISC-V-based SoC and thoroughly evaluated our prototype in terms of hardware and performance overhead. CURE imposes a geometric mean performance overhead of 15.33% on standard benchmarks.

1 Introduction

For decades, software attacks on modern computer systems have been a persisting challenge leading to a continuous arms race between attacks and defenses. The ongoing discovery of exploitable bugs in the large code bases of commodity operating systems have proven them unsuitable for reliable protection of sensitive services [104, 105]. This motivated various hardware-assisted security architectures integrating *hardware security primitives* tightly into the System-on-Chip (SoC). Capability-based systems, such as CHERI [100], CODOMs [95], IMIX [30], or HDFI [82], offer fine-grained protection through (in-process) sandboxing, however, they cannot protect against privileged software adversaries (e.g., a malicious OS). In contrast, security architectures providing Trusted Execution Environments (TEE) enable isolated containers, also called *enclaves*. Enclaves allow for a coarse-grained but strong protection against adversaries in privileged software layers. TEE architectures have been proposed for a variety of computing platforms[1], in particular for modern high-performance computer systems, e.g., industry solutions like Intel SGX [35], AMD SEV [38], ARM TrustZone [3], or academic solutions such as Sanctum [22], Sanctuary [10], Keystone [48], or Komodo [27] to name some.

In this paper, we focus on TEE architectures for modern high-performance computer systems. We investigate the shortcomings of existing TEE architectures and propose an enhanced and significantly more flexible TEE architecture with a prototype implementation for the open RISC-V architecture.

Deficiencies of existing TEE architectures. So far, existing TEE architectures have adopted a *one-size-fits-all* enclave approach. They provide only one *type* of enclave requiring applications and services to be adapted to these enclaves' features and limitations, e.g., Intel SGX restricts system calls of its enclaves and thus, applications need to be modified when being ported to SGX which produces additional costs. Additional efforts like Microsoft's Haven framework [5] or Graphene [87] are needed to deploy unmodified applications to SGX enclaves. Moreover, today, we are using diverse

[1] TEE architectures for resource-constrained embedded systems (e.g., Sancus [66], TyTAN [8], TrustLite [47] or TIMBER-V [98]) are not the subject of this paper.

services that process sensitive data, e.g., payment, biometric authentication, smart contracts, speech processing, Machine Learning as a Service (MLaaS), and many more. Each service imposes a different set of requirements on the underlying TEE architecture. One important requirement concerns the ability to securely connect to devices. For example on mobile devices, privacy-sensitive data is constantly collected over various sensors, e.g., audio [9], video [83], or biometric data [19]. On cloud servers, massive amounts of sensitive data are aggregated and used to train proprietary machine learning models, often outside of the CPU, offloaded to hardware accelerators [84]. However, TEE architectures such as SGX [35], SEV [38] and Sanctum [22], do not consider secure I/O at all, solutions such as Keystone [48] would require additional hardware to support DMA-capable peripherals, solutions like Graviton [96] require hardware changes at the peripheral side. TrustZone [3], Sanctuary [10] and Komodo [27] cannot bind peripherals directly to individual enclaves.

Another important requirement imposed on TEE architectures is an adequate and practical protection against side-channel attacks, e.g., cache [11,50] or controlled side-channel attacks [65,92,101]. Current TEE architectures either do not include cache side-channel attacks in their threat model, like SGX [35], or TrustZone [3], only provide impractical solutions which heavily influence the OS, like Sanctum [22], or do not consider controlled side-channel attacks, e.g., SEV [38]. We will elaborate on the related work and the problems of existing TEE architectures in detail in Section 9.

This work. In this paper, we present a TEE architecture, coined CURE, that tackles the problems of existing solutions with a cost-effective and architecture-agnostic design. CURE offers multiple types of enclaves: (i) sub-space enclaves that isolate only parts of an execution context, (ii) user-space enclaves, which are tightly integrated into the operating system, and (iii) self-sustained enclaves, which can span multiple CPU-cores and privilege levels. Thus, CURE is the first TEE architecture offering a high degree of freedom in adjusting enclave boundaries to fulfill the individual functionality and security requirements of modern sensitive services such as MLaaS. CURE can bind peripherals, with and without DMA support, exclusively to individual enclaves. Further, it provides side-channel protection via flexible and fine-grained cache resource allocation.

Challenges. Building a TEE architecture with the described properties comes with a number of challenges. (i) New hardware security primitives must be developed that allow enclaves to adapt to different functionality and security requirements. (ii) Even though the security primitives should allow flexible enclaves, they must not require invasive hardware modification, which would impede cross-platform adoption. (iii) While the changes in hardware should remain small, performance overhead for managing enclaves in software must be minimized. (iv) Protections against the emerging threat of microarchitectural attacks in form of side-channel and transient-execution attacks must be considered in the design for all types of enclaves.

Contributions. Our design of CURE and its implementation on the RISC-V platform tackles all these challenges. To summarize, our main contributions are as follows:

- We present CURE, our novel architecture-agnostic design for a flexible TEE architecture which can protect unmodified sensitive services in multiple enclave types, ranging from enclaves in user space, over sub-space enclaves, to self-contained (multi-core) enclaves which include privileged software levels and support enclave-to-peripheral binding.
- We introduce novel hardware security primitives for the CPU cores, system bus and shared cache, requiring minimal and non-invasive hardware modifications.
- We prototype CURE for the open RISC-V platform using the open-source Rocket Chip generator [4].
- We evaluate CURE's hardware and software components in terms of added logic and lines of code, and CURE's performance overhead on an FPGA and cycle-accurate simulator setup using micro- and macrobenchmarks.

2 System Assumptions

CURE targets a modern high-performance multi-core system, with common performance optimizations like data and instruction caches, a Translation Lookaside Buffer (TLB), shared caches, branch predictors, respective instructions to flush the core-exclusive resources, and a central system bus that connects the CPU with the main memory (over a dedicated memory controller) and various peripherals.

System bus and peripherals. The system bus connects the CPU to a plethora of system peripherals over a fixed set of hardwired peripheral controllers. The peripherals range from storage, communication, and input devices to specialized compute units, e.g., hardware accelerators [37]. The CPU interacts with peripherals using parts of the internal peripheral memory which are mapped to the address space of the CPU, called Memory-Mapped I/O (MMIO). We assume that the CPU can nullify the internal memory of a peripheral to sanitize its state. Every access from the CPU to a peripheral is decoded in the system bus and delegated to the corresponding peripheral. The CPU acts as a *parent* on the system bus, whereas the peripherals (and main memory) act as *childs* that respond to requests from a parent. However, MMIO is not sufficient for some peripherals where large amounts of data need to be shared with the CPU since the CPU needs to copy the data from the main memory to the peripheral memory. Therefore, these peripherals are often connected to the system bus as *parents* over Direct Memory Access (DMA) controllers, allowing them to directly access the main memory. To cope with resource contention in these complex interconnects, system buses also incorporate arbitration mechanisms to schedule the

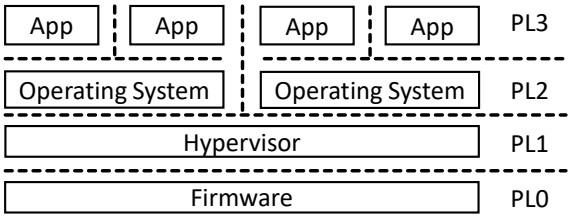

Figure 1: Software privilege levels (PL): user space, kernel space & dedicated levels for hypervisor & firmware.

establishment of parent-child connections when multiple bus requests occur simultaneously.

Software privilege levels. We assume the CPU supports the privilege levels (PLs) as shown in Figure 1. In line with modern processors (Intel [21], AMD [34] or ARM [55]), we assume a separation between a user-space layer (PL3) and a more privileged kernel-space layer (PL2), which is performed by the MMU (configured by PL2 software) through virtual address spaces. The CPU may support a distinct layer for hypervisor software (PL1) to run virtualized OS in Virtual Machines (VMs), where the separation to PL2 is performed by a second level of hardware-assisted address translation [73]. Lastly, we assume a highly-privileged layer (PL0) which contains firmware that performs specific tasks, e.g., hardware emulation or power management.

We assume that the system performs secure boot on reset, whereas the first bootloader stored in CPU Ready-Only Memory (ROM), verifies the firmware through a chain of trust [53]. After verification, the firmware starts execution from a predefined address in the firmware code and loads the current firmware state from non-volatile memory (NVM) where it is stored encrypted, integrity- and rollback-protected. The cryptographic keys to decrypt and verify the firmware state are passed by the bootloader which loads the firmware into Random-access Memory (RAM). Rollback protection can be achieved, e.g., by making use of non-volatile memory with Replay Protected Memory Block (RPMB) partitions or by using eFuses as secure monotonic counters [56]. When a system shutdown is performed, the firmware stores its state in the NVM, encrypted and integrity- and rollback-protected.

3 Adversary Model

Our adversary model adheres to the one commonly assumed for TEE architectures, i.e., a strong software-only adversary that can compromise all software components, including the OS, except a small software/microcode Trusted Computing Base (TCB) which configures the hardware security primitives of the system, manages the enclaves and which is inherently trusted [3, 10, 22, 27, 35, 48].

We assume that the goal of the adversary is to leak secret information from the TCB or from a victim enclave. An adversary with full control of the system software can inject own code into the kernel (PL2) and even into the hypervisor (PL1). This allows the adversary, with full access to the TCB interface used for setting up enclaves, to spawn malicious processes and even enclaves. Even though the adversary cannot change the firmware code (which uses secure boot), memory corruption vulnerabilities might still be present in the code and be exploitable by the adversary [24]. In addition, we assume that an adversary is able to compromise peripherals from software to perform DMA attacks [63, 76].

We assume the underlying hardware to be correct and trusted, and hence, exclude attacks that exploit hardware flaws [40, 86]. We also do not assume physical access, and thus, fault injection attacks [6], physical side-channel attacks [46, 62] or the physical connection of malicious peripherals are out of scope. We do not consider Denial-of-Service (DoS) attacks in which the adversary starves an enclave since an adversary with control over the OS can shut down the complete system trivially. As standard for TEE architectures, CURE does not protect from software-exploitable vulnerabilities in the enclave code but prevents their exploitation from compromising the complete system.

4 Requirements Analysis

To provide customizable, practical and strongly-isolated enclaves, CURE must fulfill a number of security and functionality requirements. We list them in the following section, and show in Section 7 how CURE fulfills the security requirements. In Section 6 and Section 8, we demonstrate how the functionality requirements are met.

4.1 Security Requirements (SR)

SR.1: Enclave protection. Enclave code must be integrity-protected when at rest, and inaccessible for an adversary when executed. All sensitive enclave data must remain confidential and integrity-protected at all times. An enclave must be protected from adversaries on all software layers (PL3-PL0), other potentially malicious enclaves, and DMA attacks [63, 76].

SR.2: Hardware security primitives. The protection of the enclaves must be enforced by secure hardware components which can only be configured by the software TCB.

SR.3: Minimal software TCB. The TCB must be protected from adversaries in all software layers (PL3-PL0) and minimal in size to be formally verifiable, i.e., a few KLOCs [44].

SR.4: Side-channel attack resilience. Mitigations against the most relevant software side-channel attacks must be available, namely, side-channel attacks on cache resources [31, 50, 70, 102], controlled side-channel attacks [65, 92, 101] and transient-execution attacks [12, 14, 43, 45, 78, 89, 90, 93].

4.2 Functionality Requirements (FR)

FR.1: Dynamic enclave boundaries. The trust boundaries of an enclave must be freely configurable such that enclaves

at different privilege levels can be supported.

FR.2: Enclave-to-peripheral binding. Secure communication between enclaves and selected system peripherals, e.g., when offloading sensitive machine learning tasks to hardware accelerators [84], must be explicitly supported.

FR.3: Minimal hardware changes. The hardware changes required to integrate the proposed security primitives into a commodity SoC (cf. Section 2) must be minimal, no invasive changes to CPU internals must be required to enable a higher adoption of CURE in future platforms.

FR.4: Reasonable performance overhead. The performance overhead incurred during enclave setup and run time must be minimized and must not render the computer system impractical for certain uses cases or degrade user experience.

FR.5: Configurable protection mechanisms. Protection mechanisms against cache side-channel attacks must be applicable dynamically at run time and on a per-enclave basis.

5 Design of the CURE Architecture

CURE provides a novel design that addresses the requirements described above and provides a TEE architecture with strongly-isolated and highly customizable enclaves, which can be adapted to the requirements of the services they protect. Unlike other TEE architectures, which only provide a single enclave-type, CURE allows to freely define enclave boundaries and thus, different enclaves can be constructed, as shown in Figure 2. First, in Section 5.1, we describe the ecosystem around CURE. Then, we elaborate on the different enclave types in Section 5.2. CURE's key component enabling this flexible enclave construction is its enclave ID-based access control in the system bus which manages all per-enclave resource mappings, e.g, peripherals or main memory, indicated by the different background patterns in Figure 2 and Figure 3. Our hardware primitives are presented in Section 5.3.

5.1 CURE Ecosystem

The ecosystem around CURE consists of device vendors which produce the devices implementing CURE, device users and service providers. Some services contain sensitive data (from the users and/or the service provider) and thus, must be protected. In CURE, sensitive services are either split into a sensitive and a non-sensitive part, which get included into an enclave and an user-space app (called host app), respectively, or alternatively, integrated entirely into an enclave, requiring only minimal modifications at the service. In the later case, the host app is only needed to trigger the enclave. Initially, the enclave binary does not contain sensitive data.

For every enclave, the service provider creates a configuration file which contains the enclave's requirements regarding system resources (e.g., memory, caches or peripherals), a version number and an enclave label L_{encl}. Enclave binary, configuration file and host app are bundled and deployed by the service provider over an app store (e.g., Google Play Store)

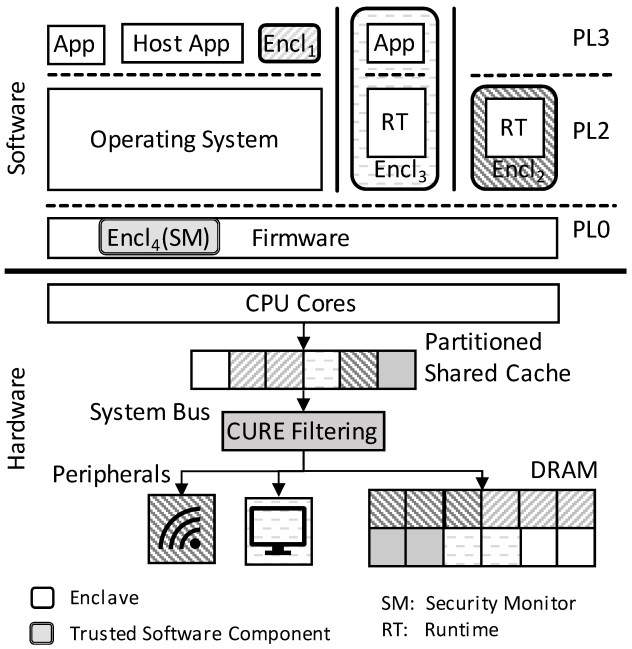

Figure 2: CURE privilege levels and enclave types, namely, user-space enclaves ($Encl_1$), kernel-space enclaves ($Encl_2$, $Encl_3$) and sub-space enclaves ($Encl_4$).

which is operated by a third party (e.g., Google). The label L_{encl} is globally unique in the app store.

Every service provider creates an asymmetric key pair SK_p and PK_p, and a public key certificate $Cert_p$, which is signed by the app store operator. Using the secret key SK_p, the service provider signs the enclave binary and configuration file (Sig_{encl}) and attaches it, together with $Cert_p$, to the app bundle. $Cert_p$ can later be used on the device to verify Sig_{encl}. For this, a certificate chain $Chain_p$ up to the root certificate of the app store operator must be present on the device. When the service provider wants to update an enclave, a new signature must be created and the version number in the configuration file updated which prevents rollbacks to older (possibly flawed) versions of an enclave [103].

A device vendor creates a unique asymmetric key pair SK_d and PK_d for each device, which is provisioned to the device during production, and a public key certificate $Cert_d$ signed by the device vendor which can later be used to prove the legitimacy of the device in a remote attestation scheme. For this, the service provider must obtain a certificate chain $Chain_d$ up to the root certificate of the device vendor. When a device was compromised, $Cert_d$ can also be revoked.

5.2 Customizable and Resilient Enclaves

CURE supports enclaves that protect user-space processes ($Encl_1$), run in the kernel space ($Encl_2$) or span the kernel and user space ($Encl_3$). However, an enclave does not necessarily include all code of a privilege level, e.g., an enclave can only comprise parts of the firmware code ($Encl_4$).

5.2.1 Enclave Management

Before describing the different enclave types supported by CURE, we give an overview on CURE's enclave management.

Security monitor. All CURE enclaves are managed by the software TCB, called *Security Monitor (SM)*, as in other TEE architectures [22, 48]. As indicated in Figure 2, the SM itself represents an enclave which is part of the firmware. As described in Section 2, we assume a system that performs a secure boot on reset, verifies the firmware (including the SM) and then jumps to the entry point of the SM. Further, we assume that the SM has already loaded its rollback protected state S_{sm} into the volatile main memory. The SM state contains SK_d, PK_d, $Cert_d$, $Chain_p$ and a structure D_{encl} for each enclave installed on the device.

Enclave installation. When an enclave is deployed to the device, the SM first verifies the signature Sig_{encl} using $Cert_p$ and $Chain_p$. Then, the SM creates a new enclave meta-data structure D_{encl} and stores L_{encl}, Sig_{encl} and $Cert_p$ in it. Moreover, the SM creates an enclave state structure S_{encl} which is used to persistently store all sensitive enclave data. The SM also creates an authenticated encryption key K_{encl} which is used to protect the enclave state when it is stored to disk or flash memory. K_{encl} and S_{encl} are also stored in D_{encl}. Initially, S_{encl} only contains an authenticated encryption key K_{com} created by the SM, which is used by the enclave to encrypt and integrity protect data communicated to the untrusted OS, and a monotonic counter. The enclave meta-data structure D_{encl} also contains a monotonic counter used to rollback protect the enclave state.

Enclave setup & teardown. The setup of an enclave is always triggered by the corresponding host app. After the OS loads the enclave binary and configuration file, it performs a context switch to the SM. The SM identifies the enclave by the label L_{encl} and begins the enclave setup by (1) configuring the hardware security primitives (Section 5.3) such that one or multiple continuous physical memory regions (according to the configuration file) are exclusively assigned to the enclave in order to isolate the enclave from the rest of the system software. Since the binary and configuration file are loaded from untrusted software, their integrity must always be verified using Sig_{encl} and $Cert_p$. Assigning physical memory regions is inevitable when providing enclaves which are able to execute privileged software (kernel-space enclave), since this allows the enclave to control the MMU. Thus, virtual memory cannot be used to effectively isolate the enclave. (2) After enclave verification, the SM configures the hardware primitives to assign also the rest of the system resources, e.g., cache or peripherals, to the enclave according to the configuration file. All assigned resources are also noted in D_{encl}. Moreover, the SM assigns an identifier to the enclave which is stored in D_{encl} and which is unique for every enclave currently active on the device. The SM can manage up to N (implementation defined) enclaves in parallel. We provide more details on the meaning of the enclave identifier in Section 5.3. (3) In the last step, the enclave state S_{encl} is restored, i.e., loaded from disk or flash memory, decrypted and verified using K_{encl}, and then copied to the enclave memory such that it is accessible during enclave runtime. The SM also checks that the monotonic counter in S_{encl} matches the counter stored in D_{encl}.

The SM configures all interrupts to be routed to the SM while an enclave is running. Thus, the SM fully controls the context switches into and out of an enclave. While the SM is executed, all interrupts on the CPU core executing the SM are disabled. All other cores remain interrupt responsive. In CURE, hardware-assisted hyperthreading is disabled during enclave execution to prevent data leakage through resources shared between the hardware threads. Alternatively, all hardware threads of a CPU core could also be assigned to the enclave if the enclave code benefits from parallelization. In the reminder of the paper, we assume that hyperthreading is disabled during enclave runtime.

After the setup is complete, the SM jumps to the entry point of the enclave. During the enclave teardown, which can be triggered by the host app or the enclave itself, the SM securely stores the enclave state (using K_{encl}), while incrementing the monotonic counters in S_{encl} and D_{encl}, removes all enclave data from the memory and caches and reconfigures the hardware primitives.

Enclave execution. At run time, enclaves can access services provided by the SM over its API, e.g., to dynamically increase the enclave's memory or to receive an integrity report which the SM creates by signing Sig_{encl} with SK_d and by attaching $Cert_d$. The integrity report is then send to the service provider by the enclave. Subsequently, using $Chain_d$, the service provider can perform a remote attestation of the enclave. Only if the attestation succeeds, the service provider provisions sensitive data to the enclave. More complex remote attestation schemes [61] could also be implemented.

Enclaves might use services of the untrusted OS which do not require access to the plain sensitive enclave data, e.g., file or network I/O. For those cases, an enclave can utilize K_{com}, which is part of S_{encl}, to protect its sensitive data. CURE also allows multiple enclaves to share encrypted sensitive data over the OS. However, the required key exchange is assumed to be performed over the back ends of the service providers and thus, out-of-scope for CURE.

Every enclave which includes a cryptographic library can also create own keys (apart from K_{com}) and store them in S_{encl}. Thus, enclaves can also implement key rotation, revocation or recovery schemes which is, however, the responsibility of the service provider and thus, out-of-scope for CURE.

On every enclave setup/teardown and context switch in and out of an enclave, the SM flushes all core-exclusive cache resources, i.e., the data cache, the TLB and the BTB, thereby preventing information leakage across execution contexts.

5.2.2 User-space Enclaves

User-space enclaves ($Encl_1$ in Figure 2) comprise a complete user-space process.

OS integration. The key characteristic of a user-space enclave is its tight integration into the OS, i.e., it relies on the OS for memory management, exception/interrupt handling and other services provided through syscalls (e.g., file system or network I/O). The OS schedules user-space enclaves like normal user-spaces processes, only that the context switches in and out of the enclave are intercepted by the SM. The OS's services are used by all user-space enclaves which prevents code duplication. Moreover, user-space enclaves do not contain management software, leading to smaller binaries.

Controlled side-channel defenses. In controlled side-channel attacks, the adversary gains information about an enclave's execution state by observing usage of resources managed by the OS, predominantly page tables [65, 92, 101]. CURE defends against these attacks by moving the page tables of user-space enclaves into the enclave memory. More subtle controlled side-channel attacks exploit the fact that the enclave's interrupt handling is performed by the OS [91]. CURE also mitigates these attacks by allowing each enclave to register trap handlers to observe its own interrupt behavior, and act accordingly if a suspicious behavior is detected [15, 79].

Limitations & usage scenarios. A user-space enclave cannot run higher-privileged code, e.g., device drivers. Thus, all sensitive data shared with a peripheral has to be processed by drivers in the untrusted OS and thus, is unprotected if not encrypted. Hence, user-space enclaves are unable to protect sensitive services which interact with devices like sensors or GPUs. Instead, user-space enclave are beneficial when protecting short-living services that can rely on encrypted data transmission, e.g., One Time Password (OTP) generators, payment services, digital key services and many more.

5.2.3 Kernel-space Enclaves

Kernel-space enclaves can comprise only the kernel space ($Encl_2$), or the kernel and user space ($Encl_3$).

Providing OS services. The key characteristic of a kernel-space enclave is its capability to run code bare-metal on a CPU core in the privileged (PL2) software layer or even in the hypervisor level (PL1) if available. Thus, OS services, e.g. memory management, can be implemented inside the enclave in a runtime (RT) component (Figure 2). This results in less resource sharing with the untrusted OS, and thus, it is easier to protect against controlled side-channel attacks [91, 92, 101]. Moreover, by including device drivers into the RT, a secure communication channel to peripherals can be established. Furthermore, kernel-space enclaves provide more computational power since CURE allows to run kernel-space enclaves across multiple cores. In CURE, peripherals can either be assigned exclusively to a single enclave, by the SM, at enclave setup or shared between different enclaves and/or the OS. The peripheral's internal memory is flushed by the SM when (re-)assigned to a new entity to prevent information leakage [49, 72, 107].

Protecting virtual machines. CURE's ability to include the kernel space into the enclave allows the construction of enclaves that encapsulate complete virtual machines (VMs). VMs are not self-contained but rely on memory and peripheral management services provided by a hypervisor, which makes the VM enclave vulnerable to controlled side-channel attacks [38, 51]. CURE mitigates this by moving the VM page tables into the enclave memory and including unmodified complete drivers into the enclave to avoid dependencies on the untrusted hypervisor [16, 17]. As for other kernel-space enclaves, peripherals are temporarily assigned to VM enclaves by the SM. Again, before a peripheral is reassigned, its internal memory is sanitized by the SM.

Limitations & usage scenarios. Sensitive services can be ported to kernel-space enclaves without changing them. However, in contrast to user-space enclaves, an enclave RT needs to be added which increases the binary size, adds development overhead and increases the memory consumption. Moreover, the CPU cores selected for the enclave first have to be freed from pending processes, detached from the OS and the RT booted on them. Nevertheless, kernel-space enclaves are required when protecting services which heavily rely on peripheral communication, e.g., authentication services using biometric sensors, ML services collecting input data over sensors or offloading computations to accelerators, DRM services or in general services which require secure I/O.

5.2.4 Sub-space Enclaves

In CURE, enclave trust boundaries can be freely defined which allows to construct fine-grained enclaves that only include parts of the software residing in a privilege level, therefore called sub-space enclaves.

Shrinking the TCB. Sub-space enclaves are especially appealing when constructed in the highest privilege level (PL0) of the system ($Encl_4$ in Figure 2). In CURE, sub-space enclaves are used to isolate the SM from the firmware code to protect against exploitable memory corruption vulnerabilities that might be present in the firmware code [24]. Moreover, hardware countermeasures, described in Section 5.3, are used to prevent the firmware code from accessing the SM data or hardware primitives. Ultimately, this minimizes the software TCB in CURE, as opposed to other TEE architectures that rely on a software TCB containing all code in the highest privilege level, i.e., EL3 (ARM) or the machine level (RISC-V), e.g., TrustZone [3], Sanctuary [10], Sanctum [22], Keystone [48].

5.3 Hardware Security Primitives

To provide CURE's customizable enclaves, new security primitives (SP) are needed in hardware. Our SPs augment the

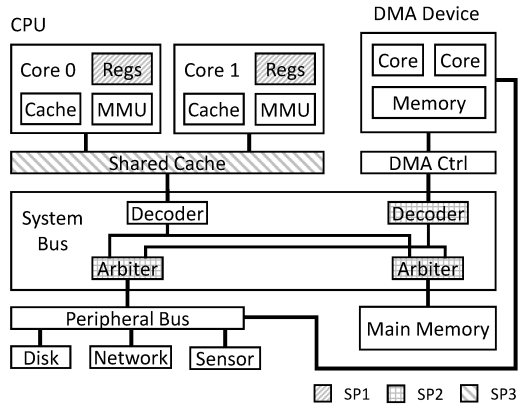

Figure 3: CURE Security Primitives (SPs), added at core register files (SP1), system bus (SP2) and shared cache (SP3).

register file of each CPU core (SP1), the system bus (SP2) and the shared cache (SP3). Figure 3 shows where CURE's SPs integrate in a modern system as assumed in Section 2.

5.3.1 Defining Enclave Execution Contexts (SP1)

Enclave ID register. In CURE, enclave execution contexts are defined using IDs, which are saved in a register that is added to every CPU core of the system (SP1). At any point in time, the value of this register, called `eid` (enclave ID) register, indicates which enclave a core currently executes. The `eid` registers are set by the SM during enclave setup, teardown and any context switch in and out of an enclave, thus, enabling flexible configuration of enclave boundaries.

Whenever an enclave is set up, the SM assigns it an unused ID. In contrast to the constant enclave labels L_{encl} (Section 5.2.1) , which are globally unique, an enclave ID is only valid as long as the enclave is loaded in memory. When an enclave is torn down, the ID gets freed and can be assigned to the next enclave. Constant IDs are only assigned to the SM and all untrusted software. The number of different IDs (N) that can be stored in `eid` defines how many enclaves can run in parallel (Section 5.2.1). However, the total number of enclaves that can be deployed is not restricted.

Propagating the enclave ID. The enclave ID is propagated through the entire system and used in the SPs to perform access control on the system resources. We incorporate the enclave ID in the bus protocol between the CPU, shared cache and system bus. In protocols like AMBA AXI4/ACE [54], the de facto on-chip communication standard, no protocol extensions are required since the bus channels provide optional user-defined signals which can be utilized to transmit the enclave ID in bus transactions. In our CURE prototype, we extend the TileLink protocol [80] by an enclave ID signal, which we describe in more detail in Section 6.

5.3.2 Access Control on the Bus (SP2)

In order to isolate enclaves and assign peripherals to them, access control mechanisms need to be implemented in hardware. As described in Section 2, the system bus represents the central gateway of a computer system that connects bus parents (CPU or DMA devices) with bus childs (peripherals or the main memory) and routes all their transactions. CURE leverages this centralization and further extends it to perform access control on parent-child transactions (SP2 in Figure 3). Incorporating carefully crafted access control at the system bus, with latency and performance in mind, reduces the overall hardware costs significantly.

Enclave memory isolation. One key task of a TEE architecture is enforcing strong isolation of the enclave code and data in the main memory. In CURE, this is achieved by performing access control in the arbiter logic in front of the main memory chip, as shown in Figure 3. This requires adding new registers and control logic to the already existing arbiter, which can only be configured (over MMIO) by the SM to assign memory regions to enclaves. Whenever the CPU requests access to a memory address, the arbiter uses the enclave ID signal, which is sent within the bus transaction, to verify if the enclave currently executing is allowed to access the memory region. At access violation, the memory access is prevented and an interrupt is triggered by the system bus, which is handled by the SM. Incorporating the required logic for this access control at the main memory side, instead of the CPU side, reduces the additional registers and logic required, which would otherwise be duplicated for every CPU core, as we show in Section 8.1.

Assigning peripherals to enclaves. The CPU interacts with peripherals over peripheral memory mapped to the CPU address space (MMIO). In CURE, access control on the MMIO memory is performed using registers and control logic added to the arbiter at the peripheral bus. The SM assigns the MMIO region of every peripheral either to one enclave exclusively or to multiple enclaves/OS by configuring the arbiter registers. Access control is then performed in the added hardware logic based on the enclave ID signal of a bus transaction. Incorporating this logic at the CPU side would have increased the hardware costs because of per-core duplication.

DMA protection. Peripherals which share large amounts of data with the CPU typically access the main memory directly over a DMA controller. CURE must protect enclaves from DMA attacks [63, 76] and also allow to assign DMA-capable peripherals to enclaves. To achieve this, CURE adds registers and control logic to the decoder in front of every DMA device. These registers define which memory regions the DMA device is allowed to access. Whenever a DMA device gets assigned to an enclave, the SM updates the device registers accordingly. Adding the required logic at the child arbiters would increase the hardware costs because enclave IDs would also need to be assigned to the DMA devices which would result in additional logic for ID comparison.

By assigning dedicated memory regions to an enclave and a DMA-capable peripheral, and by assigning the MMIO memory regions of that peripheral exclusively to the enclave, CURE

achieves an enclave-to-peripheral binding. Since neither the OS nor any other enclave can access the memory regions over which the bound enclave and peripheral communicate, no encryption or authentication schemes are required.

5.3.3 On-Demand Cache Partitioning (SP3)

CURE's enclave management (in Section 5.2.1) mitigates side-channel attacks on core-exclusive resources, such as the L1 cache, by flushing all such structures at every enclave context switch. Nevertheless, this still leaves enclaves vulnerable to cross-core attacks on the shared last-level cache [36, 39, 102]. However, vulnerability to these sophisticated attacks depends on whether the enclave code performs memory accesses dependent on sensitive data. While algorithms and implementations can be constructed leakage-resilient [2, 68], this is not directly applicable to any given application code, and thus, we provide on-demand per-enclave cache partitioning in CURE.

Security guarantees for cache side-channel resilience can be provided in hardware by either enforcing strict partitioning of resources across the different enclaves [42, 58, 97] or deploying randomization-based cache schemes [59, 60]. Nevertheless, these schemes either reduce the cache resources available for an enclave or incur additional access latency. This results in an inevitable performance overhead on the protected as well as unprotected software. The additional security guarantee, along with its resulting performance cost, is not usually required for all enclaves and largely depends on the use case.

Thus, CURE addresses these diverse enclave requirements and incorporates on-demand way-based partitioning of the shared cache (SP3 in Figure 3). This allows that cache partitioning is enabled and configured individually and dynamically for each enclave at setup and runtime. Each cache way can be allocated exclusively to an enclave. Access control on the enclave ID signal of the memory access transaction is used to permit the enclave to access (read/write or even evict) a cache way, thus ensuring strict isolation. However, when this cache isolation is not enabled for an enclave, only read/write access control on the owner enclave of each cache line is performed. This defends against a privileged adversary that can access cached enclave memory by mapping it into its own address space. As each cache line is owned by a single enclave at any point in time, access control on cache lines corresponding to shared memory between enclaves and the OS is a challenge. To address this, the SM flushes relevant cache lines at context switches between an enclave and the OS while managing shared-memory communication.

We deploy way-based partitioning because it is the least extensive in terms of hardware modifications. However, CURE provides the necessary infrastructure and mechanisms (by identifying each enclave and propagating this throughout the system bus and shared cache) to incorporate more sophisticated side-channel-resilient cache designs [25, 74, 99].

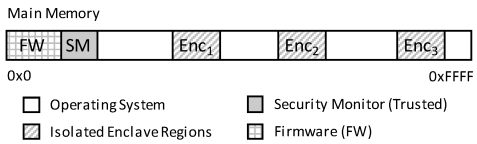

Figure 4: Physical memory layout of our CURE prototype.

6 Prototyping CURE on RISC-V

While CURE is architecture-agnostic and can be ported to other ISAs, we prototype it here for a RISC-V system based on the open-source Rocket Chip generator [4]. We describe next our CURE instantiation, followed by details on the implemented enclave types and hardware security primitives.

RISC-V System-on-Chip platform. We build a RISC-V System-on-Chip (SoC) using the Rocket Chip generator [4]. For prototyping, we equipped the SoC with multiple in-order Rocket cores, in line with prototyping efforts in related work [22]. Each Rocket core has one hart (representing a hardware thread), an own MMU, BTB, TLB and L1 cache. The SoC also contains a system bus which connects the cores to system peripherals (over the peripheral bus) and system main memory. We integrate a shared L2 cache [81] between the system bus and the main memory. A DMA device is connected to the system bus as a bus parent. As a result, this SoC resembles our assumed platform shown in Figure 3, except that the L2 cache is integrated as a last-level cache after the system bus.

We implement our prototype on this SoC aiming to maintain minimum hardware and no additional latency. We use 4 bits to represent the enclave ID, i.e., our prototype can distinguish 16 (N) enclaves, where ID `0` is statically assigned to the OS, ID `0xF` to the Security Monitor (SM) and ID `0xE` to the firmware (explained in Section 6.2.2). The remaining 13 IDs can be freely assigned to enclaves. We assign one continuous physical memory region to each enclave, resulting in the memory layout shown in Figure 4. We choose to assign only one region per enclave to simplify our prototype and minimize the induced hardware overhead. The CURE design, however, also allows for multiple continuous regions per enclave. The SM and firmware memory regions are adjacent since they are both deployed as part of the bootloader [29]. All regions not assigned to an enclave, SM or the firmware, belong to the OS. Supporting more enclaves in parallel is possible if the additional hardware overhead is acceptable.

Software stack. The Rocket core supports three software privilege levels (user, supervisor and machine). Hypervisor support is still a work-in-progress [28] and thus, we do not consider it in our prototype. In the supervisor level, we use an OS consisting of a modified Linux LTS kernel 4.19 with a Busybox 1.29.3 environment. We add a custom kernel module which performs security-uncritical tasks during the enclave setup. We implement the SM in the machine level as a sub-space enclave to separate it from the firmware which runs in the same privilege level.

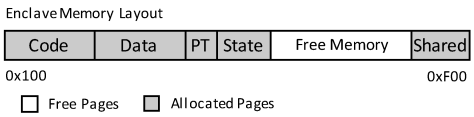

Figure 5: CURE enclave memory layout consisting of the code & data pages, page tables (PT), the enclave state (State) and the shared memory (Shared).

Cryptographic underpinnings. In the implemented CURE prototype, we use Ed25519 [71] as the digital signature scheme for the signing and verification of the enclave signature Sig_{encl} and the integrity report used for remote attestation, as described in Section 5.2.1. Thus, SK_d/PK_d and SK_p/PK_p are Ed25519 key pairs. The public key certificates $Cert_d$ and $Cert_p$ are implemented in the X.509 format. In our CURE prototype, the certificate chains $Chain_d$ and $Chain_p$ required to verify $Cert_d$ and $Cert_p$ are, for the sake of simplicity, represented by two Ed25519 public keys. As described in Section 5.2.1, $Chain_p$ is included in the SM, whereas $Chain_d$ is required at the service provider. The enclave state S_{encl} and enclave data communicated with the OS are protected through authenticated encryption, using the keys K_{encl} and K_{com}, respectively. We use AES-GCM from libtomcrypt 1.18.2. [52] as the authenticated encryption scheme and include it in the SM. Moreover, we also add it to our implemented enclaves, such that the enclaves can create additional keys. Consistent with Section 5.2.1, the SM holds a metadata structure D_{encl} for each enclave which contains $Cert_p$, Sig_{encl}, K_{encl} and S_{encl}, whereas K_{com} is part of S_{encl}.

6.1 Software CURE Enclaves

Our CURE prototype implements user-space enclaves, kernel-space enclaves and sub-space enclaves and thus, fulfills requirement `FR.1` (Section 4.2). In the following, we describe the enclave memory layout and give implementation details on each enclave type.

6.1.1 Enclave Memory Layout

In our prototype, each enclave is assigned a continuous physical memory region which is allocated during enclave setup using Linux's Contiguous Memory Allocator (CMA). The enclave memory layout is shown in Figure 5. At the lowest address, the enclave code and data pages are loaded by the OS. The enclave page tables are only stored in the enclave memory while the memory management is performed by the untrusted OS. During the enclave setup, the SM loads the enclave state S_{encl} into the enclave memory. The free memory space is used for dynamic memory allocation. The memory region at the highest address is used for the communication between enclave and OS. Since our prototype allows one continuous memory region per enclave, the shared memory region is either assigned to the communicating enclave or to no enclave, which automatically assigns the region to the OS. When the enclave is set up, the address of the shared memory region is communicated to the OS via the return value of the SM call. The enclave is informed by storing the address information on the stack of the enclave. The size of the enclave state and shared region can be freely set, we set them to 64 bytes and 4 KB, respectively.

6.1.2 Security Monitor

We implement the SM as a sub-space enclave (Enc_5 in Figure 2) separated from the firmware in memory (Figure 4), which is enforced by the hardware security primitives. However, this leaves the firmware with access to the security-critical machine level registers `eid`, which we added, and `mtvec`, which holds the base address of the trap vector that the core jumps to after an interrupt. To prevent the firmware from configuring these registers, we implement a hardware mechanism that ensures that the `eid` and `mtvec` registers can only be written to when the `eid` register is set to the SM ID (`0xF`). The `eid` register is, in turn, set to `0xF` by the hardware when performing a context switch to machine mode that traps in the SM.

6.1.3 User-space Enclaves

Memory management. Since the memory management of the user-space enclave (Enc_1 in Figure 2) is performed by the untrusted OS, we include the enclave page tables in the enclave memory, to prevent page table based attacks [65, 92, 101]. During enclave setup, the OS creates the page tables exactly as for a normal process. However, the OS turns off demand paging and maps all code and data pages to prevent page faults during enclave execution. The page tables are then handed to the SM which verifies their validity. Moreover, the SM verifies that the supverisor address translation and protection (`satp`) register, which holds the address of the root page table, points into the enclave memory. Subsequently, the page tables are copied to the enclave memory. Once the enclave is setup, the OS cannot alter the page tables anymore. When the dynamic allocation of memory leads to a page fault, the OS creates a new page table entry and passes it to the SM which includes it into the page tables.

Syscalls. Our prototype provides enclaves which can use OS services, e.g., file or network I/O, over Linux syscalls which trap in the SM. We include AES-GCM into the enclaves to encrypt and integrity-protect sensitive data shared with the OS, using K_{com}. Enclaves are always exited through the SM which is enforced by clearing the machine exception delegation (`medeleg`), machine interrupt delegation (`mideleg`), supervisor exception delegation (`sedeleg`) and supervisor interrupt delegation (`sideleg`) registers during enclave setup. During run time, the enclave can register custom trap handlers which are called by the SM before switching to the

OS after an interrupt. Thus, the enclave can observe its own interrupt behavior and detect suspicious behavior caused by interrupt-based side-channel attacks [15, 91].

6.1.4 Kernel-space Enclaves

Our CURE prototype supports kernel-space enclaves with and without user space (Enc_3 and Enc_2 in Figure 2). We use an Linux LTS kernel 4.19, which currently on RISC-V does not support a suspension mode, as the enclave RT.

Allocating resources. When an enclave is set up, the custom kernel module unmounts the driver modules of all peripherals requested by the enclave. Then, the SM performs the security-critical tasks of the enclave setup, as described in Section 5.2.3. When the enclave binary is successfully verified, the kernel module shuts down the core(s) reserved for the enclave using the Linux hotplugging mechanism. Next, a switch to the SM is performed which jumps to the entry point of the enclave RT in order to boot the RT on all reserved cores. At enclave shutdown, the SM performs the cleanup, and all freed cores are reintegrated into the OS. Then, the kernel module remounts the driver modules.

Enclave-OS communication. Since our CURE prototype allows one memory region per enclave, access to a shared region needs to be requested at the SM which then assigns the shared region to the requesting party (sender). Once the sender is finished accessing the shared region, the SM assigns the shared region to the receiver and notifies the receiver about new data in the shared region using an inter-processor interrupt. In contrast to the user-space enclave, only external interrupts are trapped in the SM during kernel-space enclave execution which is enforced by configuring the `medeleg` and `sedeleg` registers during the enclave setup. All interrupts triggered by the enclave cores are handled by the RT.

6.2 Hardware Security Primitives

We describe next, how we modify the Rocket Chip to implement CURE's hardware security primitives (Section 5.3).

6.2.1 Extending the TileLink Protocol

We modify the Rocket core such that on every memory access, the `eid` register value is sent as part of the issued bus transaction. This also includes transactions issued by the PTW (page table walker) during the page table walk when performing address translations. Thus, if a malicious enclave modified its own page tables to point to a memory region outside of the enclave memory, the PTW transactions are blocked by the access control mechanisms on the system bus.

TileLink [80] is the default bus protocol used on the Rocket Chip to connect on-chip components. TileLink specifies five channels (`A` - `E`). When connecting a parent to the system bus which contains an internal cache, all five channels are needed

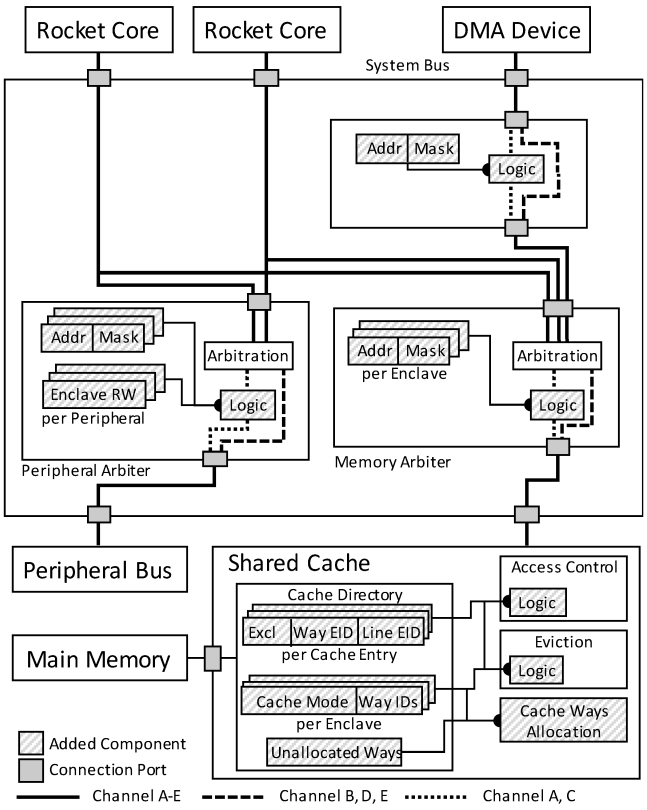

Figure 6: CURE prototype implementation using Rocket Chip.

to implement the TileLink coherence protocol (TL-C). When a parent does not require cache coherency, only the `A` and `D` channels are needed (TL-UL/UH). In our RISC-V SoC, the Rocket cores and the DMA devices are connected over TL-C since they contain L1 caches.

We extend the TileLink protocol by a 4-bit `eid` signal to propagate the enclave ID. The `eid` signal is only added to the `A` and `C` channels which transport the memory read and write transactions from the parents (CPU and DMA devices) to the system bus and childs (peripherals and main memory), respectively. All other channels remain unmodified.

6.2.2 System Bus Access Control

We implement CURE's access control mechanisms in the system bus by adding registers and control logic at the memory and peripheral arbiters and the ports connecting DMA devices. The hardware changes are shown in Figure 6, exemplary for a system containing two cores, one DMA device and multiple peripherals. All newly added components are connected to the control bus of the system and thus, are configurable by the SM over MMIO. We omit the control bus in Figure 6 for the sake of clarity. Our implementation supports enclave-to-peripheral binding and thus, fulfills `FR.4`. Moreover, in contrast to related work [20, 23], all access control is performed in parallel to arbitration, thus, guaranteeing execution in a single clock cycle without incurring additional latency.

Performing access control. The added registers hold memory ranges defined by a 32-bit base address (`Addr`) and a 32-bit mask (`Mask`), and are used by the control logic to perform access control on every memory transaction using the `eid` and `address` signals. Access control is only performed on channels with a parent-to-child direction (channels `A` and `C`). At access violation, the transaction is redirected (with all-zero data) to an unused, zero-initialized memory region. Thus, all forbidden transactions write/read zeros to/from the unused memory region. An adversary enclave might fill L1 with malicious data which could get flushed with SM privileges during enclave context switch. To prevent this, we modify the core such that on every switch to the SM, the L1 is flushed before the `eid` register is set. We connect the system bus to the peripheral and interrupt bus. This allows the SM to configure the added registers and control logic, and trigger an interrupt upon access violation which is handled by the SM.

Memory arbiter. We add 15 registers to the memory arbiter, one for each enclave (13), the SM and the firmware. Each register defines the memory region assigned to each execution context. For the enclaves, the control logic verifies that transactions only target the assigned region. For the SM, no access control is performed. The OS is allowed to access all regions except the ones specified in registers of the arbiter. The firmware is allowed to access its own and the OS regions which is why a static ID needs to be assigned to the firmware.

Peripheral arbiter. We add two registers per peripheral to the arbiter of the peripheral bus. One covers the MMIO region of the peripheral, and the other 32-bit register contains a bitmap that defines read and write permissions for every enclave.

DMA port. We add a register at every port which connects a DMA device. In CURE, a DMA device is exclusively assigned to a single enclave at one point in time. In our prototype, a DMA device accesses the main memory but not other peripherals. If specific use cases, e.g. PCI peer-to-peer transactions [67], must be supported, additional registers need to be added to specify multiple allowed memory regions. Together with the peripheral arbiter, this fulfills FR.2.

6.2.3 L2 Cache Partitioning

For cache side-channel resilience, we implement way-based flexible cache partitioning for the shared L2 (last-level) cache [81] in our prototype. We leverage the `eid`-extended TileLink memory transactions to detect when an enclave issues a cache request.

Configurable partitioning. We implement two modes of partitioning to allow enclaves to individually enable cache side-channel resilience. The first mode `CP-BASIC` performs rudimentary access control where each enclave is only permitted to access (hit) its own cache lines, but is free to evict cache lines from other ways. The second mode `CP-STRICT` provides more stringent security guarantees by allocating *exclusively* one or more ways (across all cache sets) to the pertinent enclave. Only these cache ways can be accessed by the enclave to store or evict cache lines. This provides strict isolation between the cache resources of the different enclaves, thus, effectively blocking cache side-channel leakage, but reduces the cache resources available for the enclave. Depending on the enclave service requirements, the partitioning mode can be configured by the SM independently for each enclave at setup and during the enclave lifetime, thus, fulfilling `FR.5`.

Access control. We extend each cache entry metadata with a 4-bit `line-eid` register encoding the owner enclave of the cache line, as shown in in Figure 6. We extend the cache lookup logic to generate a hit only when both tag as well as `eid` match for `CP-BASIC`, as opposed to usual tag matching.

To support `CP-STRICT`, the cache ways directory is also extended with a 1-bit register `excl` that identifies whether each way is owned exclusively by an enclave, as well as a 4-bit `eid` register that identifies the owner enclave. The cache controller logic is augmented with a register-based lookup table that is indexed by the `eid`. It encodes with a single `mode` bit whether the corresponding enclave has `CP-STRICT` enabled and its allocated cache way indices. In `CP-STRICT`, cache hits are only allowed in these cache ways.

Eviction and replacement. The L2 cache we use implements a pseudo-random replacement policy where any way is selected pseudo-randomly for eviction. We modify this to only select a way from the subset of ways allowed for each enclave. For enclaves with `CP-STRICT`, only ways exclusively allocated to it are used. For enclaves with `CP-BASIC`, all ways (except ways allocated exclusively to other enclaves) are used.

Per-enclave cache allocation. Unallocated way indices are maintained in a register vector. If an enclave with `CP-STRICT` enabled requests to exclusively own cache ways, the required ways are allocated if available and below the allowed maximum per enclave.

An inherent drawback of this partitioning technique is how the limited number of cache ways directly constrains the number of simultaneous enclaves that can have `CP-STRICT` enabled. However, this is only an implementation decision for our particular prototype, where more sophisticated cache designs [25, 74, 99] can be integrated into CURE.

7 Security Considerations

To protect from a strong software adversary, our instantiation of CURE must fulfill the security requirements introduced in Section 4.1. In the following section, we discuss how our prototype meets the requirements `SR.1`, `SR.2`, and `SR.4`, whereas we show the fulfillment of `SR.3` in Section 8.

7.1 Hardware Security Primitives (`SR.2`)

The enclave protection is enforced by hardware SPs at the system bus and L2 cache which are configured over MMIO.

After the system is powered on and on every switch to the machine level, the CPU jumps to the trap vector whose address is stored in the `mtvec` register. The trap vector is included into the SM such that the boot process and context switches are overlooked by the SM. The `mtvec` register is assigned to the SM by coupling the access permission to the SM enclave ID (stored in the `eid` register) which is also assigned to the SM. The `eid` register is set by hardware during the context switch into the machine level. During boot, the SM assigns the SP MMIO regions exclusively to its own enclave ID.

7.2 Enclave Protection (SR.1)

At rest, the enclave binaries are stored unencrypted in memory. However, during enclave setup, the SM verifies the binaries using digital signatures. Moreover, the L1 is flushed during setup/teardown to remove malicious or sensitive data from the cache. The communication between enclaves and the OS is controlled by the SM, so is the delegation of the shared memory address. Hardware-assisted hyperthreading is disabled during enclave execution. The enclave state, which is loaded during the setup process, is persistently stored by the SM using authenticated encryption, either in RAM as part of the SM state or evicted to flash/disk, and additionally rollback protected. During teardown, the SM removes all enclave data from the memory.

The SPs in hardware perform access control on physical addresses at the system bus. Thus, CURE protects from adversaries in privileged software levels (PL2 - PL0) and from off-core adversaries, e.g. peripherals performing DMA. The enclave data cached in the L1 during run time is protected by flushing it on all context switches. Data in the L2 cache is protected by assigning cache lines exclusively to enclaves. Since no enclave (except the SM), has elevated rights on the system, CURE also protects from malicious enclaves.

7.3 Side-channel Attack Resilience (SR.4)

Cache side-channel attacks. Side-channel attacks which target data in core-exclusive cache resources, i.e., in the L1 [11], the BTB [50] or the TLB [31], are prevented by the SM by flushing the resources on all context switches. Side-channel attacks targeting data in the shared L2 cache [36, 39, 102] are prevented through strict way-based cache partitioning.

Controlled side-channel attacks. Side-channel attacks on user-space enclaves which target page tables [65, 92, 101] are prevented by including the page tables into the enclave memory and by mapping all enclave code and data pages before execution. The SM verifies the page tables and the base address of the root page table stored in the `satp` register. The hardware SPs prevent the page table walker (PTW) from performing forbidden memory access during the page table walk. Side-channel attacks exploiting interrupts [91] can be mitigated using trap handlers (Section 5.2.2).

CURE provides cryptographic primitives in the user-space enclaves to encrypt and integrity-protect data shared with the OS. However, using OS services over syscalls always comprises a remaining risk of leaking meta data information [2, 77] or of receiving malicious return values from the OS [13]. In user-space enclaves, these attacks must be mitigated on the application level inside the enclave, e.g., by using data-oblivious algorithms [2, 68] or by verifying the return values [13]. None of these attacks pose a threat to kernel-space enclave since all resources are handled by the enclave RT. However, on VM enclaves, the second level page tables need to be protected, as with user-space enclaves. Interrupt-based attacks can again be mitigated with custom trap handlers. No additional countermeasures are needed to protect the SM since the SM does not use a virtual address space or OS services and handles its own interrupts.

Transient execution attacks. The discovered transient execution attacks either mistrain the branch predictor [14, 43, 45], rely on information leakage [89] or malicious injections [90] on the L1 cache, or rely on resources shared when using hardware-assisted hyperthreading [12, 78, 90, 93, 94]. By disabling hyperthreading during enclave execution (or alternatively assigning all threads to the enclave) and flushing core-exclusive caches, CURE protects enclaves against the known transient execution attacks.

8 Evaluation

In the following section, we systematically evaluate our CURE prototype. First, we quantify the software and hardware modifications required to implement CURE. Next, we evaluate the performance of CURE's enclaves using microbenchmarks, and the overall performance overhead of CURE using generic RISC-V benchmark suites.

8.1 System Modifications

Component	LOC
Linux Kernel	375 (modified)
Custom Kernel Module	200
Security Monitor	544
SM Crypto-Library	2586

Table 1: Lines of code required to implement CURE. SM Crypto-Library refers to the crypto library (part of tomcrypt) included in the Security Monitor.

Software changes and TCB. Our implementation of CURE on RISC-V comprises of a slightly modified Linux LTS kernel 4.19, a custom kernel module, and our software TCB (SM). In Table 1, the lines of code (LOC) are shown for each of the components, which indicate that the software changes required to implement CURE are minimal. Moreover, the SM only consists of around 3KLOC of code, whereas most

(82.62%) of the SM code consists of cryptographic primitives. Because of its minimal size, formal verification of the SM is possible [44], thus, fulfilling SR.3. Note that since CURE isolates the SM in an own sub-space enclave, CURE can achieve a smaller TCB size than other RISC-V security architectures [22, 48, 98] which include all code in the machine level, i.e., the firmware code, in the TCB. In our implementation, the firmware code consists of 3286 LOCs. Thus, by isolating the SM in a sub-space enclave, we managed to cut the software TCB in half, where the actual management code is even less (15.56%).

Protecting a sensitive service in a user-space enclave requires to add a small custom library (10KB) to the service binary. For the kernel-space enclaves, management code (the enclave RT) must be added in addition. In our prototype, we use the Linux LTS kernel 4.19 as the RT which increases the size of the service binary by 3MB. Custom RTs can further decrease this kernel-space enclave overhead. However, kernel-space enclaves will always have an increased binary size and memory consumption compared to user-space enclaves.

Hardware overhead. We evaluate the hardware overhead of our changes by synthesizing the generated Verilog descriptions using Xilinx Vivado tools targeting a Virtex UltraScale FPGA device. Table 2 shows a breakdown of the individual area overhead of the different modifications required to implement CURE. Overhead is represented in look-up tables (LUTs), the fundamental programmable logic blocks of FPGA devices, and registers.

Configuration	LUTs Overhead (+%)	Registers Overhead (+%)
Baseline	61,097	28,012
TileLink extension	+211 (0.4%)	+110 (0.4%)
Access control extensions		
Main memory	+5,276 (8.6%)	+1,055 (3.8%)
1 MMIO peripheral	+248 (0.4%)	+107 (0.4%)
1 DMA device	+112 (0.2%)	+72 (0.3%)
On-demand cache partitioning		
w/ L2 cache (baseline)	+30,232	+11,549
w/ L2 cache partitioned	+516 (1.7%*)	+214 (1.8%*)

Table 2: Hardware overhead breakdown in LUTs and registers. Baseline setup consists of 2 Rocket cores without L2 cache. *Overhead relative to the L2 cache (baseline).

We compare in Table 2 with a baseline configuration of 2 in-order Rocket cores (each with L1 cache). Extending the TileLink protocol throughout the system bus incurs a minimal overhead of 105 LUTs per core relative to the baseline (211 LUTs for 2 cores). This overhead includes propagating the `eid` in tandem with memory access transactions through the MMU of every core, and is thus replicated for every additional core in the system.

In contrast, the rest of our modifications for performing access control at the system bus, including enclave-to-peripheral binding, are independent of the number of cores. Incorporating logic to perform access control for every MMIO peripheral utilizes an additional 248 LUTs, and 112 LUTs per DMA device. Each represent below 0.5% overhead relative to a dual-core baseline SoC. Integrating an L2 cache into our baseline setup utilizes an additional 30,232 LUTs. Applying our on-demand way-based partitioning to this cache costs only 516 LUTs and 214 registers, which is 1.8% overhead relative to the L2 cache logic utilization itself, and 0.5% relative to the entire SoC. Our area overhead evaluation results demonstrate that the hardware modifications required to achieve our fine-grained and customized enclave protection in CURE indeed incur minimal area overhead on both single- and multi-core architectures, thus fulfilling FR.3.

Measurement	Normal Process	User-Space Enclave	Kernel-Space Enclave
Setup:	0.741	23.918	413.726
Binary Verification	-	21.824	218.975
Others	0.741	2.094	194.750
Teardown:	0.065	23.531	103.517
Memory Cleaning	-	9.384	50.206
Others	0.065	14.147	53.311
Context switch to OS	0.008	0.025	53.308
Context switch from OS	0.078	0.084	194.747
Dynamic memory allocation	0.003	0.020	0.005
OS communication	-	0.020	0.049

Table 3: CURE performance overhead compared to a normal process on microbenchmarks in milliseconds.

8.2 Performance Evaluation

We evaluate the performance of CURE using our FPGA-based setup coupled with cycle-accurate simulators. We conduct our experiments using micro and macro benchmarks for user-space and kernel-space enclaves, and compare them to unmodified user-space processes. We conduct 10 runs for each of the experiments.

8.2.1 Microbenchmarks

For microbenchmarks (Table 3), we measured important key aspects individually: setting up and tearing down an enclave, context switching with the OS, dynamic memory allocation, and communication via shared memory. We implement an application which performs the required tasks (without any additional logic) and run it as a normal Linux process, a user-space enclave and a kernel-space enclave (single core). The enclave setup is triggered by a host app in Linux which is the only purpose of the app. The enclave binary sizes therefore mainly correspond to the overhead produced by the enclave types, i.e., 10KB for the user-space enclave and around 3MB for the kernel-space enclave.

For the enclave setup, our results show that most of the time (91.3% for user-space, 52.1% for kernel-space enclaves) is spent on binary verification. The *Others* measurement

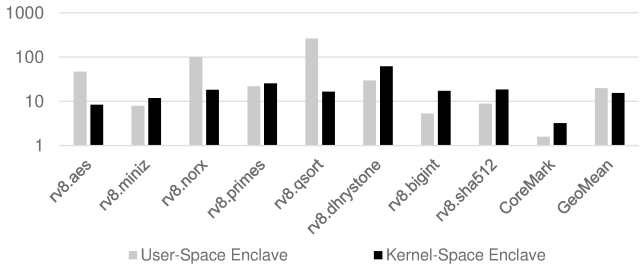

Figure 7: CURE performance overhead (in percent) on macro benchmarks `rv8` and `CoreMark` relative to a normal process.

contains all remaining steps of the setup process, e.g., loading of the enclave binary, enclave configuration, flushing of the TLB and L1 cache and jumping into the enclave. During our evaluation, we use 32KB 8-way set associative L1 data and instructions caches and a TLB with 32 entries. The setup of the kernel-space enclave is more complex and includes additional setup steps, namely, freeing the core from pending processes, detaching the core from the OS, and booting the RT. In the teardown phase, zeroing the memory produces 39.9% of the overhead for the user-space and 45,7% of the overhead for the kernel-space enclave). The cleaning is more time consuming for the kernel-space enclave because of the larger enclave memory region. The *Others* measurement contains additional steps, e.g., exiting the enclave and flushing the TLB and L1 cache. In the kernel-space enclave case, the core must additionally be rebooted.

As the RT in our prototype does not support a suspension mode (keeping the enclave in memory), we emulate the *context switch to the OS* by performing a teardown without zeroing memory, and the *context switch from the OS* by performing a setup phase without verifying the enclave binary. Suspending the enclave and restoring it should be faster than a regular shutdown and boot, thus, this represents a worst-case approximation. The context switching measurements also contain the overhead for flushing the TLB and L1 cache, for which we measure 28 cycles and 3141 cycles, respectively.

As new entries to the page tables need to be verified by the SM, user-space enclaves have a higher overhead for dynamic memory allocation. In the kernel-space enclave case, all page tables are included in the enclave memory and thus, do not need to be verified. During communication, the OS can directly access a process's memory, whereas the user-space enclave needs to copy the data to be shared to the shared memory region. The kernel-space enclave additionally has to request the shared memory from the SM, and the OS needs to be notified by the SM using an inter-process interrupt.

8.2.2 Macrobenchmarks

To evaluate the performance overhead in realistic scenarios, we used three different benchmarking suites that stress single cores, multi-core setups with two cores under test, and how the enclaves influence an OS under load. Furthermore, we measure the performance impact of our L2 cache partitioning by assigning 1/16 of the L2 cache to the enclave under test.

Single-core benchmarks. For single-core performance, we evaluated CURE with the RISC-V benchmark suites `rv8` [75] and `CoreMark` [26], which are commonly used for TEE architectures [22, 48]. The results depicted in Figure 7 are normalized to a normal user-space process. We measured a geometric mean of 19.70% for user-space enclaves and 15.33% for kernel-space enclaves for the performance overhead. As shown in Table 3, kernels-space enclaves have an increased setup time which however, amortizes with longer enclave run times. Outliers like `aes`, `norx` and `qsort` are memory-intensive workloads that perform a large number of context switches to the OS, mainly for dynamic memory allocation. Performing context switches and dynamic memory allocation is more expensive for the user-space enclave since the SM must verify newly created page table entries and copy them to the enclave memory. During one run, we count 24,601 syscalls for `aes`, 24,602 syscalls for `norx` and 48,846 syscalls for `qsort`. We also measure the overhead for flushing the TLB and L1 on every context switch which is, however, only necessary for the user-space enclave. The flushing induces only a small overhead which makes up for 1.03%, 1.48% and 1.21% of the overall overhead for `aes`, `norx` and `qsort`, respectively.

Load/Cores	Normal Process	Kernel-Space Enclave
30/1	1.49	1.49 (+-0.00%)
30/2	0.75	0.78 (+4.00%)
500/1	27.65	28.82 (+4.23%)
500/2	14.42	14.60 (+1.25%)
1000/1	56.00	55.28 (-1.29%)
1000/2	27.64	27.81 (+0.62%)
1500/1	83.62	83.64 (+0.02%)
1500/2	41.82	42.62 (+1.91%)
2000/1	111.70	111.99 (+0.26%)
2000/2	56.00	57.62 (+2.89%)
GeoMean	-	+0.9%

Table 4: Kernel-space enclave performance on multi-core `stress-ng` benchmark in seconds.

Multi-core benchmarks. Since CURE allows to assign multiple core to a kernel-space enclave, we evaluated CURE also on the dedicated multi-core benchmark `stress-ng` [41]. The results in Table 4 show that multi-core kernel-space enclaves are practical by achieving almost the same performance as normal processes.

Influence on OS. We stress the OS by running `CoreMark`, while starting an enclave in parallel. For the user-space enclave we use a single core, while two cores are needed for the kernel-space enclave, for which we simulate the suspension mode as in the microbenchmarks. For one core, the CoreMark running on the OS is slowed down by 0.519s (1.56%). For two cores with only one call after setting up the kernel-space enclave, the OS is slowed down by 0.792s (4.23%), showing

Benchmark	Cycles # for 16/16 ways (baseline)	Cycles # for 1/16 ways (worst-case)	Overhead (+%)
rv8.aes	29,754,631,670	32,175,733,155	8.1%
rv8.miniz	42,040,536,353	45,063,752,315	7.2%
rv8.norx	30,899,386,564	32,702,249,193	5.8%
rv8.primes	21,731,621,683	21,770,731,965	0.18%
rv8.qsort	24,355,792,115	25,280,228,818	3.8%
rv8.dhrystone	19,865,586,529	20,289,555,571	2.1%
rv8.bigint	65,512,466,917	71,487,944,568	9.1%
CoreMark	394,664,199	402,293,814	1.9%
GeoMean	-	-	3.09%

Table 5: Performance impact of L2 cache strict way-based partitioning for kernel-space enclaves on different benchmarks.

that the kernel-space enclave has a higher performance impact on the OS than the user-space enclave. Based on these results, we demonstrate that CURE also fulfills FR.4 and achieves a moderate performance overhead.

L2 cache partitioning. We evaluate the performance impact of partitioning the L2 cache (`CP-STRICT` mode) for kernel-space enclaves and show our results in Table 5. For our cycle-accurate experiments, we configure the core with 64KB 8-way set-associative L1 data and instructions caches and 2048KB 16-way set-associative shared L2 cache. The impact of way-based cache partitioning on performance is very application-dependent (besides the caches configuration and caches and main memory access latencies), as demonstrated by our experiments where the performance overhead ranges from a little under 0.2%, as for the `prime` benchmark, to a little over 9% for the `bigint` benchmark, for example. We measure a geometric mean of 3.09%. We note that the overheads reported are performance hits where the baseline is a best-case scenario where the only workload utilizing the cache resources (all 16 ways of the L2 cache) is the kernel-space enclave under test. Furthermore, we observe that performance significantly improves once more than 1 way is allocated per enclave, which is the likely scenario for enclaves that run applications with larger working sets and can benefit more from increased L2 cache resources.

9 Related Work

The existing works mostly related to CURE are TEE architectures which focus on modern high-performance computer systems. In contrast to capability systems or memory tagging extensions [30, 82, 88, 95, 100], TEE architectures protect sensitive services in security contexts (enclaves) against privileged software adversaries. We do not further discuss TEE architectures focusing on embedded systems [8, 47, 66, 98].

We compare CURE to other TEE architectures in Table 6. All presented architectures provide a single type of enclave which, on an abstract level, resemble either the user-space or kernel-space enclaves provided by CURE.

Intel SGX [64] offers user-space enclaves on Intel processors. The untrusted OS provides memory management and other OS services, e.g. exception handling, to the enclaves. SGX does not protect against cache side-channel [11, 50] and controlled side-channel attacks [91, 92, 101]. Many extensions to SGX were proposed in order to mitigate side-channel attacks [1, 2, 7, 15, 69, 79], however, these solutions are all ad-hoc approaches that do not fix the underlying design shortcomings of SGX, but instead leverage costly data-oblivious algorithms [1, 2, 7], or exploit not commonly available hardware in an unintended way [15, 79].

Sanctum [22], which also provides user-space enclaves, addresses both, cache side-channels through page coloring, and controlled side-channels by storing the enclave page tables in the enclave memory, like CURE. However, page coloring is not practical as it influences the whole OS memory layout and cannot be efficiently changed at run time. CURE's cache partitioning instead allows dynamic assignment of cache ways, and also mechanisms to mitigate interrupt-based side-channel attacks. Sanctum and SGX only provide user-space enclaves which are inherently limited as they cannot provide secure I/O, but only protect from simple DMA attacks.

Similar to SGX, AMD SEV [38], which isolates complete VMs in the form of kernel-space enclaves, does not consider any side-channel attacks. VM data in the CPU cache is protected by an access control mechanism relying on Address Space Identifiers which, however, does not protect against cache side-channel attacks. As the memory management and I/O services are provided by the untrusted hypervisor, SEV is also vulnerable to controlled side-channel attacks [65] and cannot provide secure peripheral binding [51].

ARM TrustZone [3] separates the system into normal and secure world, a single kernel-space enclave which does not rely on the OS and thus, is protected from controlled side-channel attacks. TrustZone does not provide cache side-channels protection, only by using additional hardware [106]. Further, TrustZone's major design shortcoming is providing only a single enclave, thus, sensitive services cannot be strongly isolated with TrustZone, hence, access to TrustZone is highly limited in practice by device vendors. Extensions building upon TrustZone mostly tried to enable multi-enclave support for TrustZone [10, 18, 33, 85] with workarounds that either rely on ARM IP [10], block the hypervisor [18, 33], or massively impact performance [85]. Since multiple enclaves were not considered in the TrustZone design from the beginning, even the proposed extensions cannot provide binding peripherals directly and exclusively to single enclaves.

Keystone [48] provides kernel-space enclaves on RISC-V. Moreover, Keystone uses a cache-way based partitioning against cache side-channel attacks, comparable to CURE. However, Keystone provides a coarse-grained cache ways assignment per CPU core, whereas CURE assigns cache ways to enclaves with freely configurable boundaries. Thus, the Keystone design is limited to a single enclave type which prevents Keystone from isolating the firmware from the actual TCB and demands adapting the sensitive services to the

Name	Extensions	Enclave Type			Dynamic Cache Side-Channel Resilience	Controlled Side-Channel Resilience	Enclave-to-Peripheral Binding
		User-Space	Kernel-Space	Sub-Space			
SGX [64]	[1, 2, 7, 15, 69, 79]	●	○	○	◐*	◐*	○
Sanctum [22]	-	●	○	○	◐	●	○
SEV(-ES) [38]	-	○	●	○	○	○	○
TrustZone [3]	[10, 18, 27, 32, 33, 57, 85, 106]	○	●	○	◐*	●	◐
Keystone [48]	-	○	●	○	●	●	○
CURE	-	●	●	●	●	●	●

Table 6: Comparison of major TEE architectures with respect to provided enclave types, dyn. cache-side channel and controlled-side channel resilience, and enclave-to-peripheral binding, i.e., MMIO/DMA protection with exclusive enclave assignment. ● indicates full support, ◐ for support with limitations, ○ for no support, * if resilience can only be achieved through extensions.

predefined enclave. Moreover, in contrast to CURE, Keystone does not support enclave-to-peripheral binding.

10 Conclusion

We presented CURE, a novel TEE architecture which provides strongly-isolated enclaves that can be adapted to the functionality and security requirements of the sensitive services which they protect. CURE offers different types of enclaves, ranging from sub-space enclaves, over user-space enclaves, to self-sustained kernel-space enclaves which can execute privileged software. CURE's protection mechanisms are based on new hardware security primitives on the system bus, the shared cache and the CPU. We instantiate CURE on a RISC-V system. The evaluation of our prototype indicates minimal hardware overhead for the security primitives and a moderate overall performance overhead.

Acknowledgments

We thank our anonymous reviewers for their valuable and constructive feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297. Moreover, this project has received funding from Huawei within the OpenS3 lab.

References

[1] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *NDSS*, 2019.

[2] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[3] ARM Limited. Security technology: building a secure system using TrustZone technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2008.

[4] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *TOCS*, 33(3):1–26, 2015.

[6] I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In *CRYPTO*, pages 131–146. Springer, 2000.

[7] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A. Sadeghi. Dr. sgx: automated and adjustable side-channel protection for sgx using data location randomization. In *ACSAC*, pages 788–800, 2019.

[8] F. Brasser, B. El Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *DAC*, pages 1–6. IEEE, 2015.

[9] F. Brasser, T. Frassetto, K. Riedhammer, A. Sadeghi, T. Schneider, and C. Weinert. Voiceguard: Secure and private speech processing. In *Interspeech*, pages 1303–1307, 2018.

[10] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *NDSS*, 2019.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *WOOT*, 2017.

[12] C. Canella, D. Genkin, L. Giner, D. Gruss, et al. Fallout: Leaking data on meltdown-resistant cpus. In *CCS*, pages 769–784, 2019.

[13] S. Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *ASPLOS*, volume 13, pages 253–264, 2013.

[14] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *EuroS&P*, pages 142–157. IEEE, 2019.

[15] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Asia CCS*, pages 7–18. ACM, 2017.

[16] H. D. Chirammal, P. Mukhedkar, and A. Vettathu. *Mastering KVM virtualization*. Packt Publishing Ltd, 2016.

[17] D. Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

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

[19] K. Choi, K. Toh, and H. Byun. Realtime training on mobile devices for face recognition applications. *Pattern recognition*, 44(2):386–400, 2011.

[20] J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar. Seca: security-enhanced communication architecture. In *CASES*, pages 78–89. ACM, 2005.

[21] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2019.

[22] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[23] P. Cotret, J. Crenne, G. Gogniat, and J. Diguet. Bus-based mpsoc security through communication protection: A latency-efficient alternative. In *FCCM*, pages 200–207. IEEE, 2012.

[24] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, pages 463–478, 2013.

[25] G. Dessouky, T. Frassetto, and A. Sadeghi. Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security*, 2020.

[26] EMBC. Coremark. https://www.eembc.org/coremark/, 2019.

[27] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*, pages 287–305. ACM, 2017.

[28] RISC-V Foundation. The risc-v instruction set manual, volume ii: Privileged architecture. https://riscv.org/specifications/privileged-isa/, 2019.

[29] RISC-V Foundation. Risc-v proxy kernel and boot loader. https://github.com/riscv/riscv-pk, 2019.

[30] T. Frassetto, P. Jauernig, C. Liebchen, and A. Sadeghi. Imix: In-process memory isolation extension. In *USENIX Security*, pages 83–97, 2018.

[31] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *USENIX Security*, pages 955–972, 2018.

[32] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *MobiSys*, pages 488–501. ACM, 2017.

[33] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing arm trustzone. In *USENIX Security)*, 2017.

[34] Advanced Micro Devices Inc. Amd64 architecture programmer's manual volume 2: System programming. https://www.amd.com/system/files/TechDocs/24593.pdf, 2019.

[35] Intel. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[36] G. Irazoqui, T. Eisenbarth, and B. Sunar. S $ a: A shared cache attack that works across cores and defies vm sandboxing and its application to aes. In *S&P*, pages 591–604. IEEE, 2015.

[37] N. P. Jouppi, C. Young, N. Patil, and D. Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, 2018.

[38] D. Kaplan, J. Powell, and T. Woller. Amd memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[39] M. Kayaalp, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. A high-resolution side-channel attack on last-level cache. In *DAC*, page 72. ACM, 2016.

[40] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[41] C. King. stress-ng. https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html, 2019.

[42] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *MICRO*, pages 974–987. IEEE, 2018.

[43] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[44] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, et al. sel4: Formal verification of an os kernel. In *SOSP*, pages 207–220. ACM, 2009.

[45] P. Kocher, J. Horn, A. Fogh, D. Genkin, et al. Spectre attacks: Exploiting speculative execution. In *S&P*, pages 1–19. IEEE, 2019.

[46] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113. Springer, 1996.

[47] P. Koeberl, S. Schulz, A. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *EuroSys*, page 10. ACM, 2014.

[48] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović. Keystone: A framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[49] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting gpu vulnerabilities. In *S&P*, pages 19–33. IEEE, 2014.

[50] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, pages 557–574, 2017.

[51] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *USENIX Security*, pages 1257–1272, 2019.

[52] LibTom. Libtomcrypt. https://www.libtom.net/LibTomCrypt/, 2019.

[53] ARM Limited. Trusted board boot requirements client (tbbr-client) armv8-a. https://static.docs.arm.com/den0006/d/DEN0006D_Trusted_Board_Boot_Requirements.pdf?_ga=2.193628069.980937939.1583698138-225494643.1545056698, 2018.

[54] ARM Limited. Amba® axi and ace protocol specification. https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf, 2019.

[55] Arm Limited. Arm® architecture reference manual. https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf, 2019.

[56] ARM Limited. Arm platform security architecture trusted boot and firmware update. https://pages.arm.com/rs/312-SAX-488/images/DEN0072-PSA_TBFU_1.0-bet1.pdf, 2019.

[57] Linaro. Op-tee. https://www.op-tee.org/.

[58] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, pages 406–418. IEEE, 2016.

[59] F. Liu and R. B. Lee. Random fill cache architecture. In *MICRO*, pages 203–215. IEEE, 2014.

[60] F. Liu, H. Wu, K. Mai, and R. B. Lee. Newcache: Secure cache architecture thwarting cache side-channel attacks. *MICRO*, 36(5):8–16, 2016.

[61] John M. Intel software guard extensions remote attestation end-to-end example. https://software.intel.com/en-us/articles/intel-software-guard-extensions-remote-attestation-end-to-end-example, 2018.

[62] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[63] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *NDSS*, 2019.

[64] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*. ACM, 2013.

[65] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. Severed: Subverting amd's virtual machine encryption. In *EuroSec*. ACM, 2018.

[66] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, 2013.

[67] NVIDIA. Developing a linux kernel module using gpudirect rdma. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html, 2019.

[68] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pages 619–636, 2016.

[69] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *USENIX ATC*, 2018.

[70] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *RSA Conference*, 2006.

[71] Orson P. ed25519. https://github.com/orlp/ed25519, 2019.

[72] R. D. Pietro, F. Lombardi, and A. Villani. Cuda leaks: a detailed hack for cuda and a (partial) fix. *TECS*, 15(1):15, 2016.

[73] M. Portnoy. *Virtualization essentials*, volume 19. John Wiley & Sons, 2012.

[74] M. K. Qureshi. Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, pages 775–787. IEEE, 2018.

[75] RV-8. Rv8-bench. https://github.com/rv8-io/rv8-bench, 2019.

[76] F. L. Sang, V. Nicomette, and Y. Deswarte. I/o attacks in intel pc-based architectures and countermeasures. In *SysSec Workshop*, pages 19–26. IEEE, 2011.

[77] R. Schuster, V. Shmatikov, and E. Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *USENIX Security*, pages 1357–1374, 2017.

[78] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019.

[79] M. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[80] SiFive. Sifive tilelink specification. https://sifive.cdn.prismic.io/sifive%2F57f93ecf-2c42-46f7-9818-bcdd7d39400a_tilelink-spec-1.7.1.pdf, 2018.

[81] SiFive. Sifive block inclusive cache. https://github.com/sifive/block-inclusivecache-sifive, 2019.

[82] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *S&P*, pages 1–17. IEEE, 2016.

[83] M. Sonka, V. Hlavac, and R. Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[84] D. Steinkraus, I. Buck, and P. Simard. Using gpus for machine learning algorithms. In *ICDAR*, pages 1115–1120. IEEE, 2005.

[85] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *DSN*, 2015.

[86] A. Tang, S. Sethumadhavan, and S. Stolfo. Clkscrew: exposing the perils of security-oblivious energy management. In *USENIX Security*, pages 1057–1074, 2017.

[87] C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *USENIX ATC*, pages 645–658, 2017.

[88] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *USENIX Security*, pages 1221–1238, 2019.

[89] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[90] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[91] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *CCS*, pages 178–195. ACM, 2018.

[92] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, pages 1041–1056, 2017.

[93] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. Ridl: Rogue in-flight data load. *S&P*, 2019.

[94] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020.

[95] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. Codoms: Protecting software with code-centric memory domains. In *ISCA*, pages 469–480. IEEE, 2014.

[96] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *USENIX OSDI 18*, pages 681–696, 2018.

[97] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh. Secdcp: Secure dynamic cache partitioning for efficient timing channel protection. In *DAC*, pages 1–6. ACM, 2016.

[98] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A. Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[99] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. Scattercache: thwarting cache attacks via cache set randomization. In *USENIX Security*, pages 675–692, 2019.

[100] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *ISCA*, pages 457–468. IEEE, 2014.

[101] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, pages 640–656. IEEE, 2015.

[102] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, 2014.

[103] Google Projekt Zero. Trust issues: Exploiting trustzone tees. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017.

[104] Google Projekt Zero. Cve-2018-17182. https://bugs.chromium.org/p/project-zero/issues/detail?id=1664, 2018.

[105] Google Projekt Zero. Xnu: copy-on-write behavior bypass via mount of user-owned filesystem image. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2018.

[106] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *CCS*, pages 1723–1740. ACM, 2019.

[107] Z. Zhou, W. Diao, X. Liu, Z. Li, K. Zhang, and R. Liu. Vulnerable gpu memory management: towards recovering raw data from gpu. *Proceedings on Privacy Enhancing Technologies*, 2017(2):57–73, 2017.