TeeM³: Core-Independent and Cooperating Trusted Execution Environments

Nils Asmussen
nils.asmussen@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Sebastian Haas
sebastian.haas@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Carsten Weinhold
carsten.weinhold@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Nicholas Gordon
nicholas.gordon@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Stephan Gerhold
stephan@gerhold.net
TU Dresden
Dresden, Germany

Friedrich Pauls
friedrich.pauls@gmail.com
Barkhausen Institut
Dresden, Germany

Nilanjana Das
nilanjana.das@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Michael Roitzsch
michael.roitzsch@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Abstract

Trusted Execution Environments (TEEs) enable secure code execution on machines that are not fully trusted by the user who runs the workload. However, existing TEE solutions mostly target CPUs and are typically tied to one specific instruction set architecture. Although some accelerators also provide support for TEEs, this leads to multiple, different TEE implementations on the same system, increasing its complexity and trusted computing base (TCB). This challenge becomes particularly apparent when workloads span heterogeneous processing units, because the diversity of TEE implementations complicates the creation of secure communication channels between the individual TEEs.

In this paper, we present TeeM³, a trusted execution architecture with discrete, out-of-core enforcement, which is core-independent and better-suited to heterogeneous systems than existing approaches. We build upon M³ and extend both the hardware platform and operating system (OS). On the OS side, we add a root of trust (RoT) for remote attestation and a unikernel-like library for TEEs. On the hardware side, we add two lightweight isolation mechanisms to protect both TEEs and the RoT from other components in the system. Furthermore, TeeM³ inherits uniform communication channels from M³ and protects them to support communicating groups of TEEs. We show in the evaluation that our approach has low performance overhead, modest additional hardware costs, and reduces the hardware TCB by a factor of 1.8 and the software TCB by a factor of 3.42.

CCS Concepts: • **Security and privacy → Hardware security implementation**; **Trusted computing**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

Keywords: Trusted Execution Environments, Operating Systems, Hardware-Software Co-Design, Accelerators

ACM Reference Format:
Nils Asmussen, Sebastian Haas, Carsten Weinhold, Nicholas Gordon, Stephan Gerhold, Friedrich Pauls, Nilanjana Das, and Michael Roitzsch. 2026. TeeM³: Core-Independent and Cooperating Trusted Execution Environments. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779212.3790232

1 Introduction

Trusted Execution Environments (TEEs) enable secure data processing by isolating a program from all other software on a system. Through remote attestation, TEEs also provide assurances that the right program is running on this system, and not a malicious replacement. These TEE features help improve data security in the cloud. However, they are also critically-important for ensuring safe and secure operation of industrial plants or connected devices in the transportation, energy, or medical sectors. In this work, we are targeting single-board computers like industrial robots, wearable medical devices, and cars, but rule out very resource-constrained use cases such as devices with a single microcontroller and kilobytes of memory. For example, consider an Internet-of-Things (IoT) device: Sensor data is collected, then pre-processed using an accelerator, and finally sent to a cloud server through a cryptographically-secured channel.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790232

By means of attestation, the device can prove its integrity and vouch for the authenticity of the sensor data. However, this simple scenario already points to the first of three challenges when adopting TEEs in modern architectures.

First, as systems become more heterogeneous, featuring multiple core architectures and various accelerators, TEE properties must be available for all processing units. This includes the accelerator used in the aforementioned IoT device and not just the processor handling cryptography and network protocols. Unfortunately, TEE support exists only for some general-purpose CPUs [1–5] and certain accelerators (mainly in the datacenter [6–8]), leaving many components unprotected [9]. Even if all processors offered individual TEE support, workloads that span multiple hardware units must employ ad-hoc measures to secure the unprotected interfaces between the various cores and accelerators. In embedded IoT systems as in our example, I/O devices such as sensors and actuators must also be secured.

The second problem is that the individual TEE implementations that already exist are highly complex, because they are baked into already complex CPU micro-architectures [10]. As a result, they are tied to a specific instruction-set architecture (ISA), limiting choice for system designers. Additionally, these TEE solutions require security-critical firmware, which is typically closed and sometimes even hidden in processor microcode blobs [11]. These black boxes cannot be inspected or changed by the user, but are nonetheless part of the trusted computing base (TCB). Relying on multiple independent TEE variants for each type of processing unit increases overall complexity of the system even further.

Third, current TEEs are typically implemented as a separate processor mode, which makes them inherently vulnerable to side-channel issues in the processor core, as demonstrated in numerous publications [12–16]. Potential attackers are not only other applications or TEEs running on the same processor, but also the operating system (OS) or hypervisor. For example, malicious system software can use interrupts [17, 18], page faults [19], or cache misses [20] as an attack vector to exfiltrate confidential data from a TEE.

To avoid side-channel issues, enable unified TEE support, and reduce complexity, we have to move enforcement of TEE security guarantees like isolation out of the individual processing units. Rather than relying on a special processor mode, there must be a separate hardware component that can protect all processors. This out-of-core *enforcement module* enables an architecture that trivially isolates single-component TEEs, marking a significant departure from existing designs. Each such component (e.g., a core, accelerator, or I/O device) is enclosed in a *tile* that is separated from others by this enforcement module. Tile-based TEEs do not share cores with other software and are therefore immune to many side-channel attacks that plague current TEE designs [14–16, 21] (see Section 3.2 for details). However, the role of the enforcement module does not need to be limited to isolation. It can also be the architectural building block that selectively allows communication among certain tiles, resulting in *logical TEEs* that can be composed of several, heterogeneous components that cooperate with each other. Tile-based isolation of physical TEEs also reduces attestation requirements, as it allows remote verifiers to only trust the involved components with the single addition of the enforcement module. Unused components can safely be ignored, as long as the enforcement module prevents unwanted interaction with the logical TEE under consideration, thereby protecting cross-component interactions.

We implement this core-independent TEE architecture using M³ [22]. We picked this hardware and OS platform for three reasons: First, M³ uses a tile-based architecture designed for heterogeneous systems [23]. Second, it already includes a hardware component for tile isolation that can be extended into the required TEE enforcement module. Third, its microkernel-based OS supports running OS components and applications on separate tiles, mitigating side-channel attacks [24]. However, as discussed in Section 3.1, several challenges remain in turning M³ into a TEE architecture. Notably, the M³ kernel and multiple OS services are still part of the TCB for any M³ application. Stated clearly, we argue that TEE platforms with heterogeneous components must provide *core-independent* enforcement by moving these mechanisms out of the cores. We present TeeM³, a platform that realizes this goal, and make the following contributions:

- Two new hardware mechanisms provide out-of-core, processing-unit-independent TEE isolation. They protect communication channels between TEEs in TeeM³, removing the need to trust the kernel or other OS components for confidentiality and integrity (Section 3.4).
- A Unikernel-like library for minimizing the complexity of the TEE runtime (Section 3.5.1).
- An implementation of a Root-of-Trust (Section 3.5.2) that enables a measured startup process for remote attestation (Section 3.6). This RoT protects itself using the newly introduced hardware isolation mechanisms.
- An FPGA-based hardware implementation of TeeM³ with two accelerators (Section 4).
- An evaluation of the performance implications, hardware costs, and TCB size (Section 5).

2 Background

In this section, we give an overview about TEEs and explain how the M³ hardware/software co-design architecture can serve as the foundation for a heterogeneous TEE platform.

2.1 Trusted Execution Environments

A *Trusted Execution Environment (TEE)* is an execution container that resides within a computer system not controlled by the party running the code. This container provides a processor and memory for the program that runs inside.

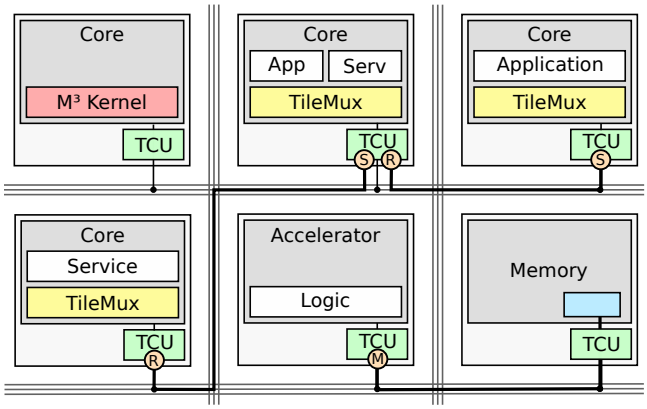

Figure 1: System architecture of M³: A TCU isolates each tile from other tiles but allows select communication as configured by the M³ kernel. TileMux allows multiple applications to share a tile.

To establish trust in the TEE, the external party requests cryptographic evidence that the TEE platform enforces specific security guarantees. This process, called *Remote Attestation*, relies on a *Root of Trust (RoT)*, typically centered on a hardware-protected secret key. Hardware and firmware components of the RoT use this key to sign evidence proving the integrity of the TEE state, such as cryptographic hashes showing that the intended program is running. The evidence also identifies the TEE hardware and firmware that enforce integrity and confidentiality.

Most TEE solutions target CPUs and assume that any software outside the TEE's isolation boundary (except TEE firmware) need not be trusted [1–4]. A consequence of this design choice is that multiple TEEs on the same system are assumed to distrust each other. This threat model is also common for accelerator-based TEEs, which often treat general-purpose processors of the host system as untrusted and instead rely on their own isolation and RoT support [6, 7, 25]. Hence, cooperation between TEEs is either not supported, or several programs running in their respective, potentially heterogeneous TEEs must establish cryptographically-protected communication channels on their own. We discuss notable (but not fully generic) exceptions [26–28] in Section 6.

2.2 M³ Hardware/Software Co-Design Platform

We use the M³ [22] hardware/software co-design platform as the basis for TeeM³. M³ builds upon a tiled hardware architecture [29] as shown in Figure 1. M³ extends each tile with a hardware component called trusted communication unit (TCU) that connects this tile to a network-on-chip (NoC). A tile contains either a general-purpose core, an accelerator, or memory. To perform message-passing or memory accesses, a corresponding *communication channel* (thick black lines in the figure) must be established. Communication channels are represented as *endpoints* in the TCU (orange circles). At runtime, each endpoint can be configured to different endpoint types: A *receive endpoint* (R) allows to receive messages, a *send endpoint* (S) allows to send messages to a specific receive endpoint, and a *memory endpoint* (M) allows a tile to issue DMA requests to tile-external memory (blue rectangle). Send and receive endpoints must be configured as pairs, whereas a memory endpoint does not require a counterpart in the memory tile's TCU.

On the software side, M³ uses a microkernel-based OS whose kernel (red) runs on a dedicated *kernel tile*. Applications and OS services run on the remaining *user tiles* and are represented as *activities*. An activity that executes code on a general-purpose tile is comparable to a process, whereas an activity on an accelerator tile uses the accelerator's logic. Both kinds of activities can use existing communication channels, but only the M³ kernel is allowed to establish such channels. By default, no communication channels exist and thus tiles are isolated from each other. An OS component called *TileMux* (yellow) allows multiple applications or OS services to share a single user tile. It runs in privileged mode of the tile's processor and takes care of isolating and scheduling the activities on this tile [30]. However, a TileMux instance has no permissions beyond its own tile. Only the M³ kernel can make system-wide decisions, hence its name.

M³ was designed for heterogeneous systems and strong isolation between tiles. The TCU in each tile offers a uniform interface for policing cross-tile collaboration. TCUs enforce access control in hardware and do not depend on any software stack on their tiles. Thus, we can easily integrate an accelerator tile that does not have a general-purpose core that could run OS-related code. Nevertheless, such tiles have full access to all OS services, because these are provided by TCU-based communication protocols.

3 Design

The overarching goal for TeeM³ is to reduce the TCB of individual TEEs, while enabling secure cooperation of multiple TEEs on the same system. M³ is a good starting point, but several challenges need to be overcome to turn it into a TEE architecture. We summarize the challenges and the threat model we derived from it, then we present the TeeM³ design.

3.1 Re-Design Challenges

C1: Kernel Privileges. The M³ kernel has access to all resources and can reconfigure all TCUs in the system. Thus, it could compromise the integrity and confidentiality of both TEEs and the RoT, if implemented as a tile. *Challenge 1: The kernel should not be able to tamper with TEEs or the RoT.*

C2: Resource Ownership. Due to M³'s hierarchical system design, parents own the memory and initial access rights used by their children. Therefore, complex services such as the pager or the file system could abuse their ownership to

compromise TEE resources. *Challenge 2: To ensure exclusive ownership, parents should not have access to TEE resources.*

C3: Resource Reclamation. We need to balance the requirement that the kernel should be removed from the TCB of a TEE with its ability to remain in control of the platform. For example, if a TEE misbehaves or the user decides to terminate it, the kernel should have a way to reclaim this TEE's resources. *Challenge 3: The kernel should be able to kill a TEE and reset its tile but not harm the TEE while it is running.*

C4: Runtime Measurement. Each processor-based M³ tile runs an instance of TileMux, which operates in privileged mode so that it can schedule multiple applications on its tile and do low-level memory management. It is required to load applications and is therefore part of the TCB for every application on that tile. Thus, TileMux would need to be measured when starting a TEE, but its internal state changes during loading, which complicates attestation. *Challenge 4: The initial state of a TEE and its runtime must be predictable at the time of measurement.*

3.2 Threat Model

We consider TCBs for the security goals confidentiality, integrity, and availability. We assume all TCUs and the NoC to be trusted, as these components are essential for the secure operation of the system. All cores and accelerators outside a TEE tile (except the RoT core), are untrusted regarding confidentiality and integrity. The RoT is part of the TCB for confidentiality and integrity, as it must attest to remote parties that these security properties hold for TEEs. However, the kernel, OS services for program loading, and the cores they run on are considered untrusted regarding confidentiality and integrity. They only remain part of the availability TCB, as they are necessary for starting applications in TEEs. All untrusted components can be under control of an attacker.

Our approach can tolerate transient-execution vulnerabilities (e.g., Spectre [31], Meltdown [21], and Fallout [32]) in cores, because each TEE has a dedicated core that does not share the resources these attacks are based on with other cores in the system. We can also tolerate isolation-breaking bugs (e.g., GhostWrite [33] and ÆPIC [34]) in cores, because TEEs are isolated via core-independent mechanisms. However, as TEEs still share the network-on-chip and DRAM, additional means are required to defend against side channels based on these resources [35, 36]. We also exclude software-exploitable vulnerabilities based on physical properties of the hardware such as power side channels [37–39] or row hammering [40]. Finally, we consider timing side-channels [41, 42] that require constant-time execution out of scope.

In contrast to typical TEE designs, we consider memory encryption and integrity protection optional. If implemented at the boundary of the system-on-chip (SoC), it provides protection against physical attacks on off-chip DRAM. If implemented within a TeeM³ SoC, it additionally removes the NoC from the confidentiality and integrity TCB. Without these features, physical attacks are considered out of scope.

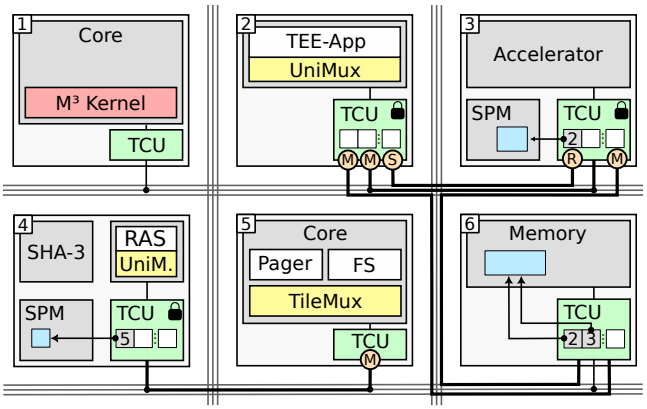

Figure 2: Architecture overview with TEE-enhanced TCUs (locking and exclusive memory regions), TEEs based on UniMux, and the new RoT tile. The example depicts the kernel tile (1), a general-purpose TEE tile (2), an accelerator TEE tile (3), the RoT tile (4), a general-purpose tile (5), and a memory tile (6).

3.3 Architecture Overview

We start with an overview of our core-independent TEE architecture by introducing the individual components and their role in the system. As illustrated in Figure 2, TeeM³ keeps the basic architecture of M³. To support stronger isolation as needed for TEEs, TeeM³ extends all TCUs with the following new isolation mechanisms:

1. **Exclusive memory regions** restrict access to specific tiles, preventing even the kernel and load infrastructure from accessing sensitive information. For example, in Figure 2, only tiles 2 and 3 have access to the blue region in memory tile 6.
2. **TCU locking** prevents changes to communication channels, ensuring that the kernel and load infrastructure cannot modify the flow of data between tiles. In Figure 2, tile 2, 3 and 4 have been locked.

These two mechanisms address **C1** and **C2** by removing the kernel and loading infrastructure from the confidentiality and integrity TCB of both the RoT and TEEs (Section 3.4.1 and 3.4.2). Through careful design, they also address **C3**. To address **C4**, we replace TileMux with a runtime component called *UniMux* that runs TEEs baremetal (tile 2 in the figure). UniMux serves a role similar to TileMux, but is stripped down and specifically built to support a single application per tile (Section 3.5.1). To support remote attestation, we added a new RoT tile to the system architecture. The RoT tile includes a SHA-3 hash accelerator, which it uses to perform a measured boot of its own firmware stages and all other tiles in the system, including the kernel tile. After booting completed, the RoT tile offers a remote-attestation service

called *RAS*. The RoT is designed to have a minimal impact on the TCB and protects itself and its secrets from the rest of the system via the TCU lock and exclusive regions (see Section 3.5.2 and Section 4.5 for details).

3.4 TCU Extensions

Turning a tile into a TEE means that everything on that tile is trusted, whereas all other tiles should be considered untrusted. As the TCU is the component that isolates a tile from the rest of the system, we enhance this hardware component such that it can enforce TEE-like isolation without relying on the OS. These TCU enhancements consist of two hardware features that (1) protect TEE memory and (2) prevent the kernel from reconfiguring TCU endpoints while the TEE is running. We designed these features carefully such that they can protect both user TEEs and the RoT. We will focus on TEEs in this section and explain how the RoT uses the new TCU protection features in Section 3.5.2.

3.4.1 Exclusive Memory Regions. In TeeM³, memory is provided either by dedicated *memory tiles* that connect to off-chip DRAM or by *compute tiles* with internal *scratchpad memory (SPM)*. DRAM regions contain code or data for compute tiles and tile-local SPM may be used by accelerators that require predictable access latency, perform many parallel memory accesses, or to store cryptographic secrets. However, applications running on other tiles may need access to certain regions of the accelerator's SPM in order to provide input data or retrieve results. In M³, access control for memory regions is controlled by the kernel, who configures memory endpoints in the "sending" TCU (i.e., the TCU of the tile that issues read or write requests). In TeeM³, we remove the kernel from the TCB of a TEE by adding support for *exclusive memory regions* in the "receiving" TCU. A TEE then needs to trust only its own tile and the TCU of the memory or compute tile that provides memory to it (**C1** and **C2**). TeeM³ supports shared access to a tile's memory for cooperating TEEs, while denying access to all other tiles.

Region Registers. Each TCU has a number of *exclusive-region registers* that can be configured with (1) an address range within tile-local memory and (2) a tile ID that specifies the owner of that region. For every incoming memory request, the TCU compares the address range of the request against its exclusive-region registers. If the request overlaps with one or more exclusive regions, the TCU will additionally compare the ID of the requesting tile against the owner tile IDs in in the matching exclusive-region registers. If the ID of the requesting tile matches the owner of at least one region, the request is permitted. Otherwise, access is denied. To keep the hardware costs low, regions are power-of-two sized and size aligned (like RISC-V's physical memory protection).

Region Management. The exclusive-region registers need to be configured by software, which needs to enforce certain policies to achieve our security goals. For example, new sharers may not be added to a region after a specific point (e.g., remote attestation) and regions must be overwritten with zeros before reusing the memory (e.g., to not leak secrets after TEE shutdown). Enforcement of such policies in hardware would be complex and inflexible. We therefore implement them in RoT firmware, which we describe in Section 3.5.2. The RoT is fully trusted for confidentiality and integrity anyway, so this decision does not weaken our trust model. To control which tile can configure exclusive-region registers, we add a new *region-manager register* to the TCU that contains a tile ID and a lock bit. The lock bit is initially disabled; once enabled, only the tile named in region-manager register's tile ID field can write to this register and the exclusive-region registers. Thus, TeeM³ is flexible enough to support a specialized region manager for each tile. However, in our prototype, we keep things simple and let RoT configure itself as the exclusive-region manager for every tile.

3.4.2 Tile Locking. In addition to enforcing access control for off-tile memory, TeeM³ must also protect individual TEE tiles. In M³, the kernel configures send and receive endpoints in TCUs to set up inter-tile message passing. However, it must be trusted to not compromise confidentiality or integrity of communication channels later on. For example, it could redirect communication channels to a different receiver or a man-in-the-middle tile that intercepts messages. TeeM³ introduces *tile locking* as a second TCU-based protection mechanism that prevents such attacks. The TeeM³ kernel still configures all send, receive, and memory endpoints in a TEE's TCU, but it cannot make arbitrary changes after the TEE started (addressing **C1** and **C2**). However, to reclaim resources of crashed or misbehaving TEEs (**C3**), the kernel can reset a tile, resulting in termination of the TEE and invalidation of all its communication channels. This mechanism is based on a lock bit and a generation counter.

Lock Bit. We added a lock bit to all TCUs in TeeM³. This bit turns a tile into a TEE by activating two protection measures: First, the kernel can no longer change TCU endpoint registers directly, but only "propose" changes that the TEE must agree to (details below). Second, locked TCUs deny all incoming memory requests unless they pass the owner check for an exclusive region as explained in Section 3.4.1. A program running in a TEE can only *enable* the TCU lock bit, but the kernel can disable it indirectly by resetting the tile. Note that resetting a tile kills the TEE, but it does not automatically remove exclusive regions owned by that TEE. Instead, the kernel must ask the RoT to remove them according to clean-up policies described in Section 3.5.2.

Frozen Endpoints. If the lock bit prevented *any* change to TCU endpoint registers, TeeM³ would impose a severe, perhaps unacceptable limitation on applications: Many M³ service protocols could no longer function and cooperating

TEEs could not dynamically establish communication channels at runtime. To make TeeM³ a practical and dynamic TEE platform (satisfying **C3**), we decided to allow changes to communication channels using a propose-freeze-accept protocol between the kernel and the TEE. When a TCU is locked and the kernel writes to an endpoint register, the TCU will *freeze* the affected endpoint. While frozen, the TEE cannot use this endpoint, but it can inspect the "proposed" change and, if deemed acceptable, *unfreeze* the endpoint with its new configuration. To do so, the TEE uses a newly-added unfreeze command of its TCU that can only be used from the TEE tile. Note that the TeeM³ kernel must be trusted not to abuse endpoint freezing for denial-of-service attacks. But since it is a microkernel, we believe that keeping it in the availability TCB is a worthwhile tradeoff.

Verification of endpoint configurations is performed in two stages. The first protects the integrity of TEEs, which only concerns receive endpoints as the only inwards facing endpoints. Without further measures, the kernel could corrupt TEE memory by overlapping a receive buffer with TEE state. Since the TEE chooses the receive-buffer address, but needs to ask the kernel for endpoint configuration, TEEs defend against this attack by checking whether exactly that address has been set in the frozen endpoint. It requires no application-specific knowledge and is done in a TeeM³-specific library. The second stage checks application-specific properties such as correct assignment of communication partners and memory areas. It is more complex and must be performed by the external verifier during remote attestation.

Tile Generations. Since the kernel is allowed to reset a (locked) tile at any time, it could kill a TEE and then put a new application at the orphaned end of an existing inter-TEE communication channel. To prevent such an attack, we introduced a *generation counter* into each TCU. This counter starts with a value of zero at boot time and is incremented every time the corresponding tile is reset. The TCU endpoints that form a communication channel between two tiles are only valid, if their own (not incrementing) generation counters match the current value of the TCU's generation counter. Otherwise, the TCU will not transmit any data and instead report a communication error, thereby preserving confidentiality and integrity of inter-TEE communication. Note that the reset operation fails, if the counter reached its maximum value; the platform must then either reboot or the RoT requires a mechanism to remove all communication channels to this tile and then reset the generation counter.

3.5 OS Extensions

Besides the hardware extensions, TeeM³ required several modifications in OS components inherited from M³. By design, M³ runs the TileMux multiplexer on all core-based tiles. Using the *multiplexer protocol*, the kernel instructs TileMux instances to start or stop activities and modify page tables. TeeM³ seamlessly integrates its RoT and TEEs into this system architecture by supporting this protocol for their tiles, too. To this end, we replace TileMux on both the RoT and TEE tiles with a less complex Rust library called *UniMux*. However, UniMux only supports one activity in a single, eagerly-mapped address space and denies all kernel requests for page-table manipulations. This design choice is intentional, as page-table updates must never be allowed for the RoT and are undesirable for user TEEs, because they complicate measuring a TEE's code and configuration state unnecessarily. TeeM³ extends the multiplexer protocol with new operations for managing exclusive regions that are only sent to and handled by the RoT.

3.5.1 TEE Runtime. To solve the self-referential problem with TileMux, whose state is changed during application load, but also needs to be measured (**C4**), we decided to sidestep this problem by running TEEs baremetal and only link them against UniMux. In contrast to VM-based TEEs, running baremetal causes no nested-paging overhead. However, running baremetal on M³ does not restrict the functionality, because all OS services are offered via TCU-based protocols, with protection of the application enforced by the TCU. By bundling TileMux as a unikernel-like library, TEEs benefit from compiler optimizations, reduced program complexity, and a smaller binary size [43, 44]. But more importantly, this approach prevents time-of-check to time-of-use (TOCTOU) attacks by using eager paging instead of demand paging. The TeeM³ kernel and loader services copy the complete binary into the exclusive TEE memory region *before* the RoT measures it. Bundling UniMux as a library also moves TileMux functionality into the attested binary, making it difficult to tamper with UniMux functionality for a particular binary. If TEEs were running on top of TileMux instead, it would be easier for the kernel to tamper with TileMux during the TEE loading process.

Note that the TeeM³ kernel and loader services including the file system need to be trusted during the initial setup. However, this "trusted during setup" approach can be turned into a "trust but verify" approach through remote attestation. By including both the hashes of the loaded software and the TCU configurations into the report, an outside party can verify that the TEE has been loaded and configured correctly.

3.5.2 Root of Trust. Our TEE architecture includes a dedicated RoT tile, which boots first and is responsible to bootstrap the kernel tile. A boot ROM in the RoT tile contains a low-complexity boot loader, which initiates a multi-stage startup process of the RoT firmware. The firmware and all its state is exclusively kept in local SPM of the RoT tile. Following the DICE [45] approach, it loads, measures, and executes later RoT firmware stages, while deriving an attestation key that will remain in SPM for use by the final firmware stage. Hash generation and symmetric key derivation uses a SHA-3 accelerator within the RoT tile (see Section 4.2) and the

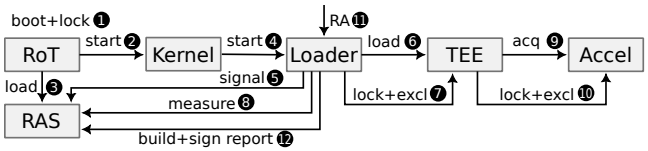

Figure 3: Steps in an end-to-end attestation scenario

`ed25519-dalek` library is used for asymmetric keys. The third stage loads and measures the code of the TeeM³ kernel and TeeM³ base services that will later be included in attestation reports. Afterwards, the third stage locks the RoT tile, starts the kernel, and replaces itself with the final RoT stage, the *remote-attestation service (RAS)*.

RAS is linked against UniMux, which starts and then waits until it receives the start-activity message from the loader. Upon receiving this message, all communication channels (e.g., to the kernel) have already been established. We chose this startup procedure to make RAS appear like an ordinary OS service, even though it is not started by the regular TeeM³ loader. Since RAS runs on a locked tile, it first verifies its TCU endpoints (e.g., making sure the receive-buffer address is as expected) and unfreezes them afterwards.

Besides creating remote-attestation reports, RAS manages the exclusive regions on all tiles (see Section 3.4.1). To support both shared memory and prevent that additional sharers are added after a certain point, we introduced the concept of *closed* regions. Until a region of a TEE is closed, overlapping regions owned by other TEEs can be added. The owning application decides, when a region should be closed. This decision is recorded by the RoT, which will then disallow any further overlaps with the closed region. During remote attestation, the verifier should accept only reports with all exclusive regions closed. RAS also validates region-removal requests by the kernel. It removes a region only if the generation counter of the owning tile is greater (i.e., after a reset) than the counter value that RAS recorded for this region. Otherwise, the removal request is denied, as the tile might still use the region. Before removing the last sharer of a region, RAS zeros the memory to ensure no secrets leak.

3.6 Putting it All Together

Now that all mechanisms are in place, we explain the individual steps that are required in an end-to-end scenario from booting the system to performing a remote attestation. These steps are illustrated in Figure 3. The system boots the RoT tile first, which runs a multi-stage firmware that measures all components and finally locks the RoT tile to protect itself (❶). Afterwards the RoT starts the kernel (❷), replaces itself with the remote-attestation service (RAS) (❸), and then waits for the start-activity signal. Like in M³, the TeeM³ kernel starts the loader on the first user tile (❹), which then starts the remaining OS services. Afterwards, the loader signals RAS to start (❺) and then loads the application onto the

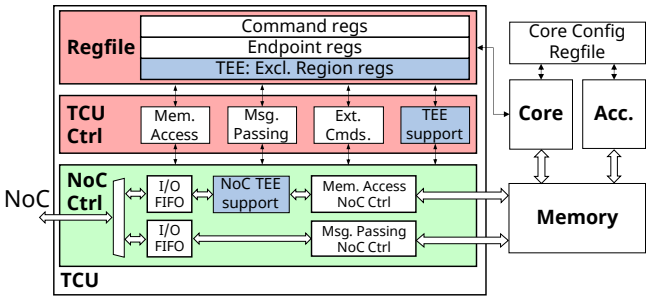

Figure 4: Architecture implementation of a hardware accelerator tile. Red-colored blocks (Regfile, TCU Ctrl) are security-relevant, while the green part (NoC Ctrl) only performs NoC transfers and memory accesses. Blue-colored boxes include hardware functionalities to support TEEs.

TEE tile (❻). The loader starts the TEE application baremetal (as it is linked with UniMux), makes the memory regions exclusive, and locks the tile (❼). Now the TEE tile and its external memory regions are protected: Other tiles (not even the kernel) cannot change its state anymore. The loader then asks RAS to measure the TEE's memory regions for a later attestation (❽). Afterwards, the TEE can start running and perform other actions, like in this case acquire the access rights for the accelerator tile (❾), lock the accelerator tile, and configure for itself (with the help of RAS) an exclusive region within the accelerator's SPM (❿). The system now runs two cooperating, heterogeneous TEEs and is ready to report its state via remote attestation. When an attestation request is received via network (⓫), the loader asks RAS to prepare and sign an attestation report (⓬). This report contains hashes over the two TEE states, all OS components loaded during boot, the system configuration, and the state of all TCUs that protect the TEEs, including their exclusive memory regions. The verifier can use this information to decide whether the system is in the expected state.

4 Implementation

We built TeeM³ upon the openly available hardware- and software platform M³ [46]. Adding support for TEEs required substantial changes to both hardware and software. We describe the most important modifications in the following.

4.1 TCU Extensions

Every tile contains a TCU, but there are different variants depending on the type of tile (e.g., memory-tile TCUs have no endpoints). Figure 4 shows the TCU components we inherited from M³ and our extensions for TEE support for an accelerator tile. The TCU consists of three functional blocks: 1) the NoC controller (NoC Ctrl) to handle data transfers to the NoC and to tile-internal components like memory, 2) the TCU controller (TCU Ctrl) that executes commands, and 3) a register file (Regfile) with command registers, through

which the tile's core can send commands to the TCU. The NoC controller is the DMA unit that handles data transfers. It operates with multiple channels to support parallel processing and to prevent deadlocks, for example, due to dependencies between accesses to the core and memory. In contrast to the TCU controller and register file, the NoC controller does not perform security-relevant tasks.

We added extensions for TEE support in three parts (blue boxes in Figure 4): First, the register file has new registers for exclusive regions (for 16 regions in the current implementation) and the region-manager register. The region-manager register stores the TCU lock bit and the number of currently configured regions. This second register field speeds up iterating through the list of active regions that must be checked for every memory request. Second, the TCU controller checks incoming memory requests and allows access according to configured exclusive regions as explained in Section 3.4.2. Since parallelizing the exclusive-region checks can improve performance, we implemented and tested multiple designs with different degrees of parallelism. We found that for our applications the best trade-off between performance and hardware resources is to check two regions in parallel. And third, the extension to the NoC controller forwards incoming data transfers to the TCU controller to check, because only the TCU controller performs such security-relevant operations. For memory read requests, the NoC controller forwards the request to the memory while letting the TCU controller perform the exclusive-region check in parallel. This may improve performance if a memory read requires a certain latency (e. g., for DRAM accesses). If the TCU controller denies the request later, read data is dropped.

4.2 SHA-3 Accelerator

We integrate a Keccak accelerator implementing SHA-3-{224, 256, 384, 512} and the extendable-output functions SHAKE-{128, 256}. The accelerator exposes a 64-bit control interface and a 128-bit memory interface backed by 4 KiB SPM. The Keccak accelerator is part of the RoT tile, which uses it to measure the software states. After booting completed, the RoT tile runs the remote-attestation service (RAS), which offers the Keccak accelerator to applications, too. Applications must configure memory endpoints in the RoT's TCU to provide it access to input data and return the hash.

4.3 AES Accelerator

We also added an Advanced Encryption Standard (AES) accelerator operating in electronic codebook (ECB) mode[1]. This simple proof-of-concept implementation encrypts one AES block (128 bits) at a time. The accelerator is integrated into a dedicated tile, which also contains a simple RISC-V core. Control firmware running on this core drives the accelerator and makes it accessible to applications. Our implementation supports the *file protocol* (introduced by M³x [23]), which enables streaming data to or from files (or UNIX-like pipes). Similarly to the Keccak accelerator, this protocol requires the input source and output sink (e.g., the file-system service) to configure a specific endpoint in the accelerator's TCU, so that the firmware on the accelerator's core can access it. This protocol is used both for input and output side, allowing the accelerator to stream data from one file, encrypt it, an then write it an output file.

[1]ECB mode is insecure and only used for simplicity.

4.4 M³ Kernel

In TeeM³, we also had to add TEE support to the kernel. In a nutshell, we extended its capability system with quota support for exclusive-region registers and added two new system calls: one for locking a tile and and another one for forwarding region-management requests to the RoT.

4.5 RoT

Besides the TCU lock to protect the RoT from the kernel, the RoT also uses an exclusive region to provide the TeeM³ loader access to a specific part of its address space. This way, the RoT can receive information from the loader without having to trust this service for any other parts of its address space. This buffer is used to communicate command-line arguments and the capability selector for the communication channel to the loader.

5 Evaluation

In the evaluation we address three questions: (1) what performance impact do the added hardware and software features incur, (2) how much do the hardware costs rise to support TEEs, and (3) how does the trusted computing base of TeeM³ compare to M³ and other TEE solutions.

5.1 Performance Implications

5.1.1 Measurement Setup. We run all of the experiments on our FPGA-based hardware implementation on an AMD Xilinx Virtex UltraScale+ FPGA (VCU118 board). The platform contains eight processing tiles with a single RISC-V core each and two memory tiles with interfaces to external DDR4 DRAM. All tiles are connected by a NoC using a 2x2 star-mesh topology. The platform contains an in-order Rocket core [47] for the kernel tile; other Rocket cores and out-of-order BOOM cores [48] are in user tiles. Both are 64-bit RISC-V cores with MMU and 16 KiB L1 cache, each for instructions and data, as well as a shared 512 KiB L2 cache. For the RoT tile and the AES tile, we use PicoRV32 [49], a tiny 32-bit RISC-V implementation. Clock frequencies are 100 MHz for the Rocket and PicoRV32 cores and 80 MHz for the BOOM cores to meet timing requirements during FPGA synthesis and place-and-route. If the standard deviation is below 1% we do not show it in the plots.

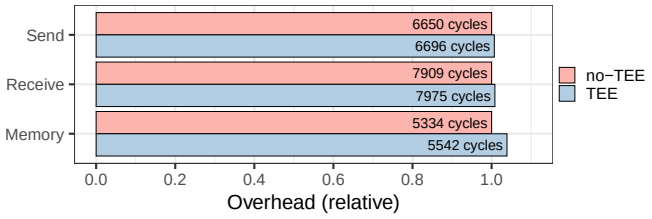

Figure 5: Communication channel creation latency as a normal application and as a TEE.

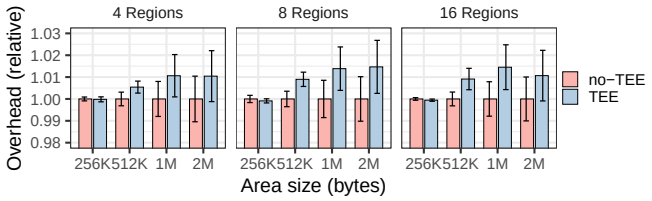

Figure 6: Overhead for DRAM accesses for TEEs (with 4, 8, and 16 exclusive regions) and non-TEEs (w/o exclusive regions) for random memory accesses within differently sized memory areas.

5.1.2 Communication Channels.

We start with micro-benchmarks that evaluate the communication performance. When running as a TEE, the tile gets locked which prevents the kernel from changing TCU endpoints (EPs) without agreement by the TEE. This causes overhead when setting up communication channels that we evaluate in this section. The benchmark creates an EP of each type with 100 repetitions after 100 warmup runs. The results in Figure 5 show that the overhead is small (below 4%), but slightly higher for memory EPs. The reason is that the kernel performs a check when creating a memory EP for an exclusive region to provide better error messages in case the tile has no access to the region. This could be optimized away, since the TCU will block access attempts anyway. The unfreeze operation itself has no significant overhead.

Note that the actual communication performance (besides setup time) is the same for ordinary applications and TEEs. This is in contrast to existing solutions like SGX, where communication with other software components is performed over the untrusted OS kernel and therefore requires enclave exits and costly protection like encryption. For example, SCONE [50] reports only 60% of the native throughput when running Redis in an SGX enclave. Solutions for VM-based TEEs like Gramine-TDX [51] must also protect all sensitive communication that leaves a confidential VM using expensive cryptographic operations. As TCU-based communication bypasses the TeeM³ kernel, no such protection is required, leading to zero overhead.

5.1.3 Exclusive Region Access.

Exclusive regions require an additional check for every memory access. We therefore evaluate the overhead when causing as many memory accesses as possible. We use two variants: 1) a cache-miss heavy workload to cause many exclusive-region checks in DRAM, and 2) TCU transfers to cause many exclusive-region checks in the SPM of another tile. For both variants, we run four instances of the benchmark, which occupies all tiles with general-purpose cores on our hardware platform. The benchmark instances are run both as normal applications and TEEs, leading only to exclusive-region checks in the latter case. Since the regions are checked in order, the time for the check depends on the number of active regions. We therefore run the benchmark with different numbers of active regions by installing an appropriate number of unused regions before the ones used by the benchmark instances.

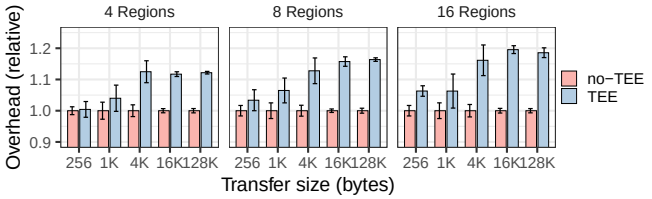

Figure 7: Overhead for SPM accesses for TEEs (with 4, 8, and 16 exclusive regions) and non-TEEs (w/o exclusive regions) for TCU-write operations with sizes.

Each benchmark instance uses 1000 and 10 repetitions for SPM and DRAM, respectively, and 10 warmup runs for both. We show the runtime and standard deviation of the worst instance in Figure 6 and Figure 7 for DRAM and SPM, respectively. As can be seen, the slowdown for DRAM accesses is below 1.5% even if all 16 exclusive regions are in use. This is because off-chip DRAM accesses are relatively slow (about 40 cycles from the memory-tile's TCU) and the exclusive-region checks for reads are performed in parallel to the actual DRAM request. In contrast, on-chip SPM accesses take a single cycle, leading to a slowdown of up to 20%.

5.1.4 Accelerator Benchmarks.

We use three different scenarios involving accelerators to demonstrate the flexibility of our approach:

- **AES stream:** The first scenario makes use of the AES accelerator to encrypt a file. The benchmark allocates the AES tile for exclusive use and loads the accelerator multiplexer onto the PicoRV32 core in that tile. Since the multiplexer understands the file protocol, the benchmark opens the input and output file and connects the multiplexer with these files. Afterwards, the benchmark lets the multiplexer stream the file data through the accelerator.
- **File hash:** The second scenario uses the SHA-3 accelerator to produce a SHA-224 hash of a file. This accelerator is part of the RoT tile and the remote-attestation service (RAS) allows applications to access it.
- **IoT MAC:** The final scenario tries to replicate a typical scenario in IoT devices: data produced by an application (e.g., read from a sensor) is stored encrypted and

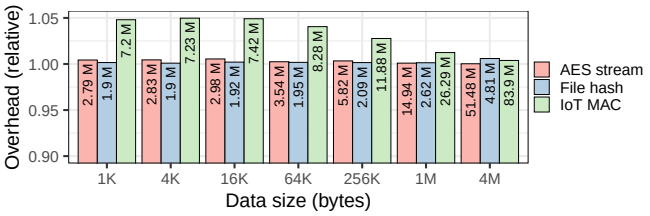

Figure 8: TEE overhead for different accelerator scenarios, depending on the data size to encrypt/hash. The number on each bar shows the total time for the non-TEE variant in cycles.

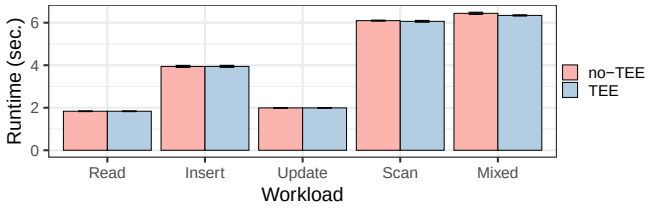

Figure 9: Runtime of different LevelDB workloads when run as a normal application ("no-TEE") and a TEE.

integrity-protected as a file. This file could be post-processed in the cloud after verifying its integrity and authenticity. We make therefore use of both accelerators to first AES encrypt the data and produce a SHA-224 hash of the encrypted data afterwards. The data is streamed through the AES accelerator via UNIX-like pipes, as already provided by M³, and also streamed to the SHA-3 accelerator via pipe.

All three scenarios are run as normal applications and as TEEs. In both cases, they are run on top of UniMux and eagerly-paged to improve the comparability. When run as a TEE, the file-hash scenario additionally runs on a locked tile and with an exclusive memory region (the RoT tile is always already locked). The AES-stream scenario additionally locks the allocated AES tile. The IoT-MAC scenario locks the AES tile as well and uses exclusive regions for the pipe buffers to protect confidentiality and integrity throughout the chain. After the benchmarks, exclusive regions are freed again. In consequence, running these scenarios with TEEs comes with more setup and tear-down overhead, but protects their execution and communication channels from other software on the system, most importantly from the kernel and load infrastructure. Note however that we do not include the zeroing of the exclusive regions in the measurements and therefore assume that sufficient unused memory is still available. Zeroing has the usual cost of a `memset` and can be avoided when using memory encryption with per-TEE encryption keys.

We run all scenarios with 8 repetitions after 2 warmup runs and vary the data size (i.e., the size of the file/data to hash/encrypt). The results are shown in Figure 8. As can be seen, the overhead is below 5% for all scenarios and is primarily caused by setup and teardown operations. For that reason, the relative overhead is larger for smaller data sizes and decreases with an increasing data size. The setup overhead is more pronounced for the IoT-MAC scenario, because it uses multiple exclusive regions for the pipe buffers, which require expensive calls to RAS on the slow RoT core.

5.1.5 Application-level Benchmark. As a final performance study, we evaluate TEE run-time overhead for a larger application scenario. The key-value store LevelDB [52] receives requests from a TCP socket, performs the requested query on its database, and then replies to the requestor. LevelDB uses an in-memory file system already provided by M³ as its backend and communicates over the loopback device of M³'s network stack with a client running on the same FPGA. We run this benchmark once with LevelDB as a TEE and once as a normal application. For comparability, LevelDB is linked against UniMux to run bare-metal in both cases.

The client uses a workload generated with the Yahoo! Cloud Serving Benchmark (YCSB) [53], which offers insert, update, read, and scan operations. As the operations stress the system in different ways, we generated workloads with different mixes: the read-heavy workload uses a proportion of 80-10-10 (80% reads, 10% inserts, and 10% updates), the insert-heavy workload uses 10-80-10, and the update-heavy workload uses 10-10-80. As scan operations are the most expensive, we handle them specially here: the scan-heavy workload uses 10% reads, 10% inserts, and 80% scans, and finally the mixed workload uses 50% reads, 30% updates, 10% inserts, and 10% scans. All workloads are generated with the Zipfian distribution and consist of 200 records that are created at the beginning, followed by 200 operations for the scan-heavy workload and 1000 operations for all others.

Figure 9 shows the result of 4 runs after 1 warmup run. As can be seen, running LevelDB in a TEE causes no performance overhead. This is because although exclusive-region checks need to be performed, the stress on the DRAM (in contrast to the worstcase analyzed in Section 5.1.3) is not sufficient to cause a measurable slowdown.

5.2 Hardware Costs

The consumed FPGA resources are listed in the first part of Table 1. The TCU variant includes all existing features including TEE support. The exclusive-region check performs two checks in parallel. Most of the resources in the TCU are spent for the I/O FIFOs in the NoC controller to ensure proper buffering and arbitration of the data between different NoC controller channels. As presented in Section 4.1, the NoC controller is the actual DMA unit to provide the interface between the NoC and the tile. These resources must always be spent in a tiled architecture even if we would remove all security features from the TCU. FPGA memory blocks are only used by the TCU's internal translation lookaside buffer to support virtual memory. Adding extensions for TEEs in

Table 1: FPGA resource consumption of the TCB in comparison to other approaches: Logic and LUT-RAM (LUTs), registers (Flip-flops, FFs), and block RAM (BR) with 36 kbit per block. For the comparison we normalize the LUTs in the last column.

Component	LUTs [k]	FFs [k]	BRs	norm.
TeeM³	**39.0**	16	1.5	1.0
├ RoT: PicoRV32+Keccak	9.6	3.9	0	
├ NoC router	3.9	2.3	0	
└ TCU	25.5	9.8	1.5	
├ NoC controller	11.4	1.8	0	
├ TCU controller	5.5	2.9	1.5	
├ Regfile	3.8	2.5	0	
└ TEE support	4.8	2.6	0	
M³ standard	**69.1**	31.3	147.5	1.8
├ RISC-V Rocket+cache	44.5	21.8	146	
├ NoC router	3.9	2.3	0	
└ TCU (w/o TEE support)	20.7	7.2	1.5	
AccShield [8]	**225.9**			5.8
CURE [54]	**97.5**			2.5
├ 2x Rocket+cache	91.7			
└ TEE support	5.8			
HECTOR-V [55]	**9.3**			0.24
├ RVSCP based on RI5CY	5.7			
├ TEE support	0.5			
└ AXI4 crossbars	3.1			

the TCU increases the number of LUTs and FFs by about 19% and 27%, respectively. The higher increase of FFs is caused by 16 additional exclusive-region registers.

Compared to the RISC-V Rocket core, the TCU takes about 57% of the LUTs. However, larger and more performant cores such as the BOOM core take up significantly more space (144.4k LUTs, 74.5k FFs, and 158 BRAMs in our setup), resulting in a relative size of 18% for the TCU. Furthermore, in contrast to the RISC-V cores, the TCU requires almost no memory resources. This is an advantage in real ASIC designs where memory area is one of the most expensive parts.

5.3 Trusted Computing Base

5.3.1 Hardware Components. We now compare the TCB for confidentiality and integrity of different approaches with TeeM³, shown in Table 1. For TEEs in TeeM³, we consider the RoT tile, the TCU, and the interconnect (now a single NoC router) as the hardware TCB. Depending on the application scenario, a TEE needs to trust other components as well, including the core it is running on. Compared to M³, whose TCB consists of the TCU, the NoC, and the kernel tile with a Rocket core, TeeM³ reduces the TCB size by a factor of 1.8.

AccShield [8] provides an FPGA prototype with TEE support for machine-learning accelerators, which are deployed in the cloud. The design contains a security manager, crypto engines, and control logic, leading to a 5.8x larger hardware TCB than TeeM³. The security architecture CURE [54] supports TEEs by modifying a CPU's register file (two Rocket cores), its L2 cache, and the system bus for access control. In this approach, the large Rocket cores remain in the hardware TCB, resulting in a 2.5x larger TCB than TeeM³.

HECTOR-V [55] implements a single hardware isolation module (as opposed to multiple identical TCUs) that enforces communication control and provides a hardened RISC-V processor (RVSCP) for TEEs. The trusted parts of a TEE for confidentiality and integrity are therefore RVSCP, the isolation module, wrappers to perform access-control checks next to each peripheral component, and the interconnect. Their peripheral wrappers require only a few FPGA resources and correspond to the TCUs in our design. However, it is still unclear how multiple TEEs could collaborate on HECTOR-V and how they can get access to OS services. As demonstrated in our performance evaluation, both processor-based TEEs and accelerators are fully integrated into the system and have direct access to OS services. TeeM³ is therefore larger, but offers a more general and flexible solution. Note however that comparisons of LUT counts, in particular for different approaches, need to be taken with a grain of salt.

Table 2: Comparison of software TCBs for confidentiality and integrity. Note (*): TeeM³ and M³ only use a fraction of these libraries.

TEE approach	SLOC	RISC-V inst.	normalized
TeeM³	< 112,824	**72,587**	1.0
├ RoT	1,824	62,751	
├ UniMux	950	9,836	
└ Libraries (*)	< 110,050		
M³ standard	< 70,588	**248,308**	3.42
├ Kernel	7,697	68,480	
├ TileMux, loader	8,188	179,828	
└ Libraries (*)	< 54,703		
Keystone	71,825	**3,776,872**	52.0
├ OpenSBI firmware	29,822	3,756,321	
├ Security monitor	6,988	13,053	
└ Eyrie runtime	35,015	7,498	

5.3.2 Software Components. Table 2 shows the code sizes of the software TCBs for TeeM³, standard M³ without TEE support, and related works. Both TeeM³ and M³ are written in Rust, with some inline assembly. We found it difficult to compare source lines of code (SLOC), because the TeeM³ RoT, kernel, and loader services have library dependencies comprising tens of thousands of lines of Rust code, of which they use only a small fraction. Precisely identifying those library functions that are actually required by the kernel, RoT, UniMux, and loader services would have required manual inspection of the whole code base. We therefore chose to compare code sizes based on the number of RISC-V instructions that end up in the individual components after the compiler and linker eliminated all unreachable library

code. Based on this metric, the confidentiality and integrity TCB for all TEEs (i.e., RoT and UniMux) is 3.42x smaller than the code size of the standard M³ kernel and loader services. The latter are still part of the TCB for availability.

Comparing TeeM³ to other solutions is difficult due to different feature sets. For example, Keystone [56] provides sealed memory, but TeeM³ does not. Nevertheless, we include code-size numbers for Keystone[2] as a reference point in Table 2. Keystone's security monitor is part of the confidentiality and integrity TCB for all TEEs. Its code is smaller than our RoT, but embedded into the OpenSBI firmware, resulting in a more than fifty times larger binary. Keystone's Eyrie runtime must be trusted by a TEE and is comparable to the binary size of UniMux.

The authors of CURE [54] report 3,130 SLOC for their security monitor, which includes cryptographic routines for signing attestation reports and is smaller than our RoT. CURE enclave applications also depend on a "small custom library (10KB)" [54] as the TEE runtime. The authors state that their security monitor is shielded from the rest of the firmware after boot, but to the best of our knowledge, the bootloader still remains part of the TCBs for confidentiality and integrity. The availability TCB of both CURE and Keystone includes an untrusted Linux OS, which is several orders of magnitude larger than the M³ kernel and loader. The publications on AccShield [8] and HECTOR-V [55] do not discuss software complexity and are therefore omitted from Table 2.

6 Related Work

TEEs for CPUs. Intel SGX [1] and TDX [2], AMD SEV-SNP [3], Arm CCA [57], or TrustZone [4] implement TEEs as a separate execution mode of a general-purpose processor. The same is true for academic works such as Sanctum [5] or Sancus [58], which add TEE execution modes to RISC-V cores or micro-controllers, respectively. This design choice makes TEEs vulnerable to side-channel attacks, as an application in a TEE can involuntarily leak its secrets to untrusted code running on the same core [14, 16, 21, 31, 32, 59, 60]. If the RoT is also integrated into such a core, micro-architectural side channels can even compromise RoT signing keys, as has been demonstrated for SGX [15]. Sanctum [5] tries to minimize this risk by implementing costly mitigations. However, research [61] indicates that such vulnerabilities cannot be ruled out completely. TeeM³ avoids side-channel attacks on TEEs and the RoT by placing each of them on a separate tile.

TEEs for Accelerators. Besides the side-channel issue, none of the aforementioned TEE solutions considers heterogeneity. Works like Graviton [6], ShEF [7], AccShield [8] and others [25, 62] provide TEE support for GPUs or other accelerators, but treat them in isolation from the rest of the system. HIX [26], SGX-FPGA [27], and MeetGo [28] go one step further by protecting device drivers for the accelerator in a CPU-based TEE, or they enable secure cooperation between CPU enclaves and an accelerator TEE. These are all specialized solutions that introduce another TEE implementation (in addition to CPU TEEs), thereby increasing the overall complexity of the system. In contrast, the enhanced TCU of TeeM³ minimizes system complexity by supporting arbitrary processing units and I/O devices with multiple, but identical instances of a single TEE enforcement mechanism.

Heterogeneous TEEs. Some works implement heterogeneous TEEs [63] using frameworks like Keystone [56] on top of CPU-based enclaves, but therefore must always trust a management CPU core that is not part of the TEE itself. SeCReT [64] highlights that inter-TEE cooperation in such a system architecture requires expensive cryptographic operations to secure the ad-hoc communication channels between them. HETEE [65] provides heterogeneous TEEs in data centers, but using a rather heavy-weight approach based on a separate PCIe-connected system running Linux as the security controller. HECTOR-V [55] and CURE [54] resemble our architecture the most. As discussed in Section 5.3.1, HECTOR-V has a smaller hardware TCB than TeeM³ at the cost of providing a less flexible solution, whereas CURE comes with a significantly larger hardware TCB than TeeM³.

7 Discussion

Generalizability. The main design principle of TeeM³ is that TEEs are isolated *out-of-core* using a hardware enforcement module that is separate from potentially complex processing units. To fully realize this design, we argue that the following three design and security concepts are necessary. First, *exclusive memory ownership* ensures that only one entity, namely the TEE, has access to its private memory. Second, TEEs have *exclusive control ownership* of their communication channels, ensuring confidentiality and integrity for all communication between cooperating TEEs. And third, a *remote system-software* design enables TEEs to use services provided by the OS, but prevents these services (including the kernel) from spying on or manipulating the TEE. None of these concepts are specific to M³, but they are all essential for out-of-core enforcement of TEE isolation.

In TeeM³, exclusive ownership for memory and communication control is enforced by two light-weight hardware mechanisms in the TCU that implement exclusive memory regions and tile locking, respectively. Both mechanisms *deprivilege* the kernel and load infrastructure such that they cannot access TEE memory, nor add or exchange communication partners. As the TeeM³ kernel and other OS services run on dedicated cores (i.e., remote to the TEE's own core), these other cores are deprivileged as well. Out-of-core TEE architectures other than TeeM³ would likely enforce exclusive

[2]Based on `keystone-enclave` master branch, commit `08a241b` as provided in latest available Docker image (2023-09-15).

memory ownership in their separate hardware enforcement modules, too. An example is CURE's memory arbiter [54], which implements physical memory protection (PMP) outside the CPU core to shield enclave memory from the untrusted OS kernel. Both CURE and HECTOR-V [55] also make a step towards securing communication channels. They feature I/O bus arbiters that protect communication between CPU-based TEEs and devices. However, they do not support exclusive control ownership for communication channels, as their "security monitors" can still make arbitrary changes to these channels. Mechanisms similar to TeeM³'s TCU lock bit and tile-generation counters would be required to prevent them from tampering with communication channels in the same way as TeeM³'s system software is prevented from manipulating endpoints in the TCU of a locked TEE tile.

TEE solutions such as Intel SGX and VM-based TEEs (e.g., TDX or SEV-SNP) also implement exclusive memory ownership, but the relevant hardware logic is tightly integrated into complex cores. Being shared-multiprocessor architectures, these platforms would require several changes in order to move enforcement of inter-core memory isolation out of their processors. A possible starting point could be the PMP-based approach demonstrated by core slicing [66], which supports *static* partitions of cores and memory regions. The resulting core-level isolation also enables a remote system-software design for the OS. Furthermore, an I/O memory management unit (IOMMU) could be enhanced to protect communication between CPU-based TEEs and accelerators, or other devices connected through an interconnect like PCI Express or AXI. A TCU-like locking mechanism and generation counters could be added to such an enhanced IOMMU, thereby enabling exclusive control ownership for *dynamic* communication channels between processor-based and device-based TEEs.

Tile-based TEEs. We deliberate chose a tile-based architecture to enforce out-of-core TEE protection, as it allows TeeM³ to isolate TEEs with any type of processing unit. However, this approach also reduces system complexity and makes it easy to reason about TEE security. Like other works [66] that do not let multiple TEEs share a core, we allow only one TEE per tile. If two TEEs *A* and *B* ran on the same tile and there was a communication channel to this tile, the TCU could not guarantee that a message for *A* will actually be delivered to *A* and not *B*. Similarly, the TCU of a memory tile could not guarantee that only *B* and not *A* has access to an exclusive-memory region. Lifting that limitation is possible, but it would make the TCB more complex. We leave the investigation of this trade-off for future work.

Message-Passing Restrictions. In contrast to memory accesses, we decided against restrictions for message passing. Since the kernel needs to be able to establish communication channels, it can also create channels that allow it (or other tiles) to send messages to all TEEs with existing receive endpoints. As all TEEs need to communicate with the kernel, it does not help to enforce that messages can only be received from specific tiles. In other words, TEEs need to be prepared for receiving messages from unexpected tiles. However, messages can easily be filtered, as the message header contains an unforgeable sender tile ID. For example, the IoT MAC scenario evaluated in Section 5.1.4 can check if data does indeed come from the sensor application. In any case, unrestricted message passing does not violate the confidentiality and integrity of TEEs, because messages are placed at receiver-controlled locations. However, it can impact the availability of TEEs, as the kernel can perform denial-of-service attacks. This is one reason why the kernel remains in the availability TCB of TEEs.

Compatibility. We provide a `musl`-based [67] POSIX emulation layer, which allows us to run existing applications with reasonable effort on TeeM³. The emulation layer is not feature complete, but supports memory management, file operations, networking, time services, and basic I/O event notification via `epoll`. Furthermore, although UniMux is written in Rust, it can be linked against both Rust and C/C++ applications to run them baremetal on a tile as a TEE. In our evaluation, we took advantage of both mechanisms to run the C++-based LevelDB as a TEE.

8 Conclusion

We present TeeM³, a new architecture for heterogeneous and cooperating trusted execution environments, implemented in the M³ hardware-software co-design. This architecture moves TEE mechanisms out of the individual cores and into a discrete hardware component, providing core-independence. By introducing exclusive memory regions and locked communication channels in this component, we protect TEEs from a potentially malicious kernel and provide a robust remote-attestation framework. After a complex software stack configures the system, the hardware component locks that configuration, allowing a remote attestation to verify it. This approach also significantly reduces the trusted computing base (TCB) required for confidentiality and integrity, leaving the operating system only in the availability TCB.

Our evaluation shows that TeeM³'s core-independence permits seamless and secure collaboration among TEEs deployed on both general-purpose and accelerator tiles. As demonstrated, adding TEE support incurs low performance overhead, modest area cost, and reduced the hardware TCB by a factor of 1.8 and the software TCB by a factor of 3.42.

Acknowledgements. We thank our shepherd, Hugo Lefeuvre, and our reviewers for their helpful suggestions. This research is funded by the European Union's Horizon Europe research and innovation program under grant agreements No. 101092598 (COREnext) and No. 101094218 (CYMEDSEC).

A Artifact Appendix

A.1 Abstract

Our work studies a hardware/operating-system co-design and therefore requires custom hardware. The artifact contains the hardware platform in form of bitfiles for the Xilinx VCU118 FPGA. For the software side, it includes the source code of all TeeM³ components and all scripts to run the benchmarks. Executing the scripts will build all required parts, run the experiments on the FPGA, and produce the raw data for the plots. For comparison, the artifact also contains the raw results used for the plots in this paper. The artifact covers all results of Section 5.1.

A.2 Artifact check-list (meta-information)

- **Program:** TeeM³ operating system, buildroot, musl, and LevelDB. All are included in the provided source code.
- **Compilation:** GCC cross compiler, Rust nightly-2024-09-04, and Vivado Lab 2019.1. The cross compiler is included and built automatically. The Rust compiler is downloaded and built automatically.
- **Hardware:** Xilinx VCU118 Evaluation Board Rev 2.0.
- **Execution:** The experiments will run for about 3 hours.
- **Metrics:** We use execution time and latency as metrics.
- **Output:** The scripts produce files with raw numbers. The expected results are included.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 3 hours
- **Publicly available?:** The source of the complete software part and the TCU and NoC of the hardware part are publicly available. The FPGA bitfiles are also publicly available.
- **Code licenses:** TeeM³ is available under GPLv2. Please refer to the LICENSE file in root directory for the licenses of the other contained components.
- **Archived (provide DOI)?:** `10.5281/zenodo.17936497`

A.3 Description

A.3.1 How to access. Both the hardware and software platform are available on Zenodo (10.5281/zenodo.17936497) and on GitHub (https://github.com/Barkhausen-Institut/M3-Bench).

A.3.2 Hardware dependencies. TeeM³ requires a custom hardware platform, which we provide in form of bitfiles for the Xilinx VCU118 FPGA.

A.3.3 Software dependencies. TeeM³ requires specific C/C++ and Rust compilers for RISC-V. These will be downloaded and built automatically. In addition, TeeM³ requires nix and Vivado Lab, which are not included in the artifact.

A.4 Installation

The repository needs to be cloned as follows:

```
git clone \
  https://github.com/Barkhausen-Institut/M3-Bench.git \
  --branch asplos26
```

The repository contains a `README.md` with all instructions to build everything and run the experiments.

A.5 Evaluation and expected results

The expected raw results are contained in the directory `expected-results` and can be compared with the raw results produced by the evaluation in the `results` directory. For example, the following commands can be used to compare the results:

```
tail -n 200 results/*.dat > res.txt
tail -n 200 expected-results/*.dat > expected.txt
vimdiff res.txt expected.txt
```

A.6 Notes

Since the measurements are done on an FPGA platform, small differences between the results reported in this paper and new runs are expected.

Note also that benchmarks will be automatically repeated on failure, because some occasional failures are unavoidable (e.g., sometimes loading programs onto the FPGA fails due to UDP packet drops). Additionally, there are still some hardware/software bugs due to system's complexity that we have not found yet.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

References

[1] Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. (Accessed on January 23, 2026).

[2] Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. (Accessed on January 23, 2026).

[3] AMD SEV-SNP: Strengthening VM Isolation With Integrity Protection and More. https://docs.amd.com/v/u/en-US/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more, 2020. (Accessed on January 23, 2026).

[4] Trustzone for Cortex A – TEE Reference Documentation. https://www.arm.com/why-arm/technologies/trustzone-for-cortex-a/tee-reference-documentation, 2021. (Accessed on January 23, 2026).

[5] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symp. (USENIX Security 16)*, pages 857–874, 2016.

[6] Volos Stavros, Vaswani Kapil, and Bruno Rodrigo. Graviton: Trusted Execution Environments on GPUs. In *Proceedings of the 13th USENIX Conf. on Operating Systems Design and Implementation*, OSDI'18, page 681–696, USA, 2018. USENIX Association.

[7] Mark Zhao, Mingyu Gao, and Christos Kozyrakis. ShEF: shielded enclaves for cloud FPGAs. In *Proceedings of the 27th ACM International*

Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22, page 1070–1085, New York, NY, USA, 2022. ACM.

[8] Wei Ren, William Kozlowski, Sandhya Koteshwara, Mengmei Ye, Hubertus Franke, and Deming Chen. AccShield: a New Trusted Execution Environment with Machine-Learning Accelerators. In *2023 60th ACM/IEEE Design Automation Conf. (DAC)*, pages 1–6, 2023.

[9] Carsten Weinhold, Nils Asmussen, Diana Göhringer, and Michael Roitzsch. Towards modular trusted execution environments. In *6th Workshop on System Software for Trusted Execution (SysTEX)*, Rome, Italy, May 2023. ACM.

[10] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel trust domain extensions (TDX) security review. *Google security review*, 2023.

[11] XuCode: An Innovative Technology for Implementing Complex Instruction Flows. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html, 2021. (Accessed on January 23, 2026).

[12] Stephan Van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *2021 IEEE Symp. on Security and Privacy (SP)*, pages 339–354. IEEE, 2021.

[13] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *International Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[14] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conf. on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 753–768. ACM, 2019.

[15] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symp. (USENIX Security 18)*, pages 991–1008, 2018.

[16] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *2019 IEEE Symp. on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[17] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conf. (USENIX ATC 17)*, pages 299–312, 2017.

[18] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. HECKLER: Breaking Confidential VMs with Malicious Interrupts. In *33rd USENIX Security Symp. (USENIX Security 24)*, pages 3459–3476, Philadelphia, PA, August 2024. USENIX Association.

[19] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symp. on Security and Privacy*, pages 640–656. IEEE, 2015.

[20] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symp. on Security and Privacy (EuroS&P)*, pages 142–157, 2019.

[21] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symp. (USENIX Security 18)*, 2018.

[22] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *21st International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 189–203. ACM, March 2016.

[23] Nils Asmussen, Michael Roitzsch, and Hermann Härtig. M³x: Autonomous accelerators via context-enabled fast-path communication. In *USENIX Annual Technical Conf. (ATC)*, pages 617–632. USENIX, July 2019.

[24] Nils Asmussen, Till Miemietz, Sebastian Haas, and Michael Roitzsch. Distrusting cores by separating computation from isolation. *Journal of Systems Architecture (JSA)*, 159, February 2025.

[25] Xingbin Wang, Rui Hou, Yifan Zhu, Jun Zhang, and Dan Meng. NPU-Fort: a secure architecture of DNN accelerator against model inversion attack. In *Proceedings of the 16th ACM International Conf. on Computing Frontiers*, CF '19, page 190–196, New York, NY, USA, 2019. ACM.

[26] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 455–468, New York, NY, USA, 2019. ACM.

[27] Ke Xia, Yukui Luo, Xiaolin Xu, and Sheng Wei. *SGX-FPGA: Trusted Execution Environment for CPU-FPGA Heterogeneous Architecture*, page 301–306. IEEE Press, 2022.

[28] Hyunyoung Oh, Kevin Nam, Seongil Jeon, Yeongpil Cho, and Yunheung Paek. MeetGo: A Trusted Execution Environment for Remote Applications on FPGA. *IEEE Access*, 9:51313–51324, 2021.

[29] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, 10 2007.

[30] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. Efficient and scalable core multiplexing with M³v. In *27th ACM International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 452–466. ACM, February 2022.

[31] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symp. on Security and Privacy (S&P'19)*, 2019.

[32] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. ACM, 2019.

[33] Fabian Thomas, Eric García Arribas, Lorenz Hetterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs. In *CCS*, 2025.

[34] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture. In *31st USENIX Security Symp.*, 2022.

[35] Antonio Savino, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Multicore dram bank-& row-conflict bomb for timing attacks in mixed-criticality systems. In *2024 IEEE 27th International Symp. on Real-Time Distributed Computing (ISORC)*, pages 1–10. IEEE, 2024.

[36] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: side-channel attacks and mitigations on mesh interconnects. In *31st USENIX Security Symp. (USENIX Security 22)*, pages 2857–2874, 2022.

[37] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel SGX. In *2020 IEEE Symp. on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1466–1482.

IEEE, 2020.

[38] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symp., USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1057–1074. USENIX Association, 2017.

[39] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking inaccessible data with software-based power side channels. In *32nd USENIX Security Symp. (USENIX Security 23)*, pages 7285–7302, 2023.

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *41st International Symp. on Computer Architecture (ISCA)*. IEEE, July 2014.

[41] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symp. on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2019.

[42] RSA Diffie-Hellman. Timing attacks on implementations of. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conf., Santa Barbara, California, USA, August 18–22, 1996, Proceedings*, volume 1109, page 104. Springer, 1996.

[43] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[44] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conf. on Computer Systems*, pages 376–394, 2021.

[45] Trusted Computing Group. Dice layering architecture, version 1.0, revision 0.19. https://trustedcomputinggroup.org/wp-content/uploads/DICE-Layering-Architecture-r19_pub.pdf. (Accessed on January 23, 2026).

[46] Microkernel-based system for heterogeneous manycores. https://github.com/Barkhausen-Institut/M3. (Accessed on January 23, 2026).

[47] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, and John Koenig. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[48] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, volume 5, pages 1–7. International Symp. on Computer Architecture Valencia, Spain, 2020.

[49] PicoRV32 - A Size-Optimized RISC-V CPU. https://github.com/YosysHQ/picorv32. (Accessed on January 23, 2026).

[50] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th USENIX Symp. on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[51] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *Proceedings of the 2024 on ACM SIGSAC Conf. on Computer and Communications Security*, CCS '24, page 4598–4612, New York, NY, USA, 2024. ACM.

[52] google/leveldb. https://github.com/google/leveldb. (Accessed on January 23, 2026).

[53] brianfrankcooper/ycsb: Yahoo! cloud serving benchmark. https://ycsb.site/. (Accessed on January 23, 2026).

[54] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symp. (USENIX Security 21)*, pages 1073–1090. USENIX Association, August 2021.

[55] Pascal Nasahl, Robert Schilling, Mario Werner, and Stefan Mangard. HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment. In *Proceedings of the 2021 ACM Asia Conf. on Computer and Communications Security*, pages 187–199, 2021.

[56] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conf. on Computer Systems (EuroSys'2020)*, pages 1–16, 2020.

[57] Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. (Accessed on January 23, 2026).

[58] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symp., Washington, DC, USA, August 14-16, 2013*, pages 479–494. USENIX Association, 2013.

[59] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[60] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symp. on Security and Privacy (S&P'20)*, 2020.

[61] Ross McIlroy, Jaroslav Sevcìk, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution, February 2019.

[62] M. E. S. Elrabaa, M. Al-Asli, and M. Abu-Amara. Secure Computing Enclaves Using FPGAs. *IEEE Transactions on Dependable and Secure Computing*, 18(2):593–604, 2021.

[63] Moritz Schneider, Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Čapkun. Composite Enclaves: Towards Disaggregated Trusted Execution. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):630–656, Nov. 2021.

[64] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent ByungHoon Kang. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *22nd Annual Network and Distributed System Security Symp., NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[65] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Lutan Zhao, Fengkai Yuan, Peinan Li, Zhongpu Wang, Boyan Zhao, et al. Enabling privacy-preserving, compute-and data-intensive computing using heterogeneous trusted execution environment. *arXiv preprint arXiv:1904.04782*, 2019.

[66] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud. In *17th USENIX Symp. on Operating Systems Design and Implementation (OSDI 23)*, pages 247–267, Boston, MA, July 2023. USENIX Association.

[67] musl libc. https://musl.libc.org/. (Accessed on January 23, 2026).