

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

CoKeMon: Configurable Kernel Monitoring by Decoupling Isolation

Clément Thorens and Shweta Shinde, *ETH Zurich*

https://www.usenix.org/conference/usenixsecurity26/presentation/thorens

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

CoKeMon: Configurable Kernel Monitoring by Decoupling Isolation

Clément Thorens
ETH Zurich

Shweta Shinde
ETH Zurich

Abstract

Monolithic OSes such as Linux are susceptible to security vulnerabilities. This has led to a line of research to detect and prevent runtime kernel exploitation, e.g., with kernel integrity measurement, page table monitoring, event logging. Prior works are either purpose-built for one type of monitoring or make invasive changes to the kernel. This motivates the need for a design that can accommodate various monitors on a need basis. As a first step, we identify five key requirements that such a solution must satisfy: selective permissions, non-privileged management, static isolation boundaries, architecturally-supported atomic switching, and device support. It is challenging to satisfy all of the requirements while preserving security. To address this, our insight is to decouple the isolation enforcement from the monitor management, which allows us to use native hardware techniques with ease. We demonstrate the feasibility of such a design on Arm platform by modifying ~300 LoC in the firmware and ~500 LoC in the Linux kernel. We implement three monitoring use-cases to demonstrate the versatility of CoKeMon and report performance overhead ranging from 0% to 13%.

1 Introduction

Linux is widely deployed across several computing platforms ranging from desktops to large-scale cloud nodes and supercomputers [53]. In particular, it has gained wide adoption in server settings [11]. Linux is also a monolithic kernel that comprises several sub-components amounting to millions of lines of code [10]. It has also been historically susceptible to a wide range of bugs (e.g., memory vulnerabilities, race conditions) [47]. A typical exploitation scenario is a remote adversary that can either run user-level programs of its choice on the platform (e.g. containers) or compromise user programs (e.g., web servers, databases, utilities) to gain access to the system [4, 6]. Such an attacker can then exploit Linux vulnerabilities to gain kernel privileges (as depicted in Figure 1a) [5, 9, 41]. There are several tools and measures to detect such bugs before they can be exploited in-the-wild to gain kernel privileges to bring about attacks (e.g., code execution, data leakage). Assuming that bugs are inevitable, it has become common to employ defenses in depth that isolate the kernel's critical components (e.g., page tables, credentials) and monitor its behavior to identify anomalies (e.g., boot and run-time integrity checks, auditing logs).

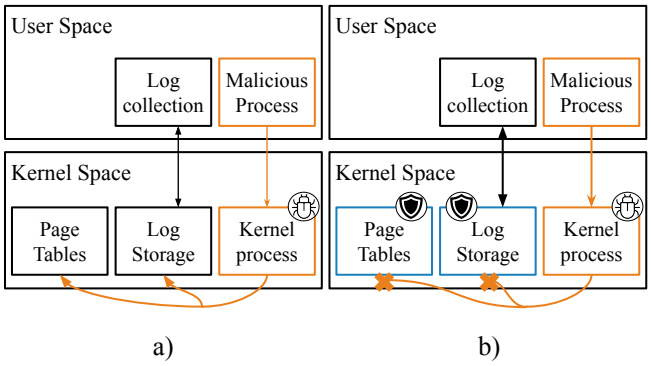


Figure 1: a) A process exploits a kernel vulnerability to compromise kernel sub-systems. b) Isolation limits the exploit.

This preventive measure stops easy exploitation of bugs. As with any second line of defense, though, the challenge shifts to ensuring that an attacker cannot trivially bypass the defense. In other words, ensuring that the monitoring itself is protected with orthogonal measures that are non-trivial for an attacker to compromise. To this end, two approaches, namely virtualization and nested kernel, are well established and principled ways to address this requirement [12, 14, 15, 18, 20, 46, 54, 57]. Both isolate the execution of privileged code that performs the critical tasks (e.g., updating data structures, monitoring/authorizing, writing logs to storage), from the rest of the kernel. The rationale is that the privileged code is small and amenable to rigorous testing or even verification, whereas the rest of the kernel code is de-privileged.

We observe that such kernel monitoring approaches either use hardware or software to create the isolation. More im-

portantly, they have to over-privilege the monitor as it is the cornerstone of the isolation. In this paper, we ask the research question: *can we decouple the role of kernel monitoring from isolation?* We answer in the affirmative and introduce a new system called CoKeMon to showcase the feasibility of such a solution. Our main insight is to rely on an isolation boundary afforded by modern hardware. In doing so, we remove the need for an over-privileged entity to do any management (e.g., changing isolation boundaries, adjusting permissions). Then, we directly leverage the context switch mechanism of the hardware to avoid complex call-gates to change privileges during execution.

CoKeMon allows execution of multiple mutually isolated monitors on the system (e.g., page table isolation and auditing). Each monitor gets a custom permissioned view of the kernel memory (e.g., no access, read-only access, access to a specific range, temporal access only before launch). Further, CoKeMon supports attaching dedicated devices to particular monitors while protecting the kernel and other monitors from device-side attacks. While these features may be easy to achieve, our choice of removing any privileged management layer makes it particularly challenging for CoKeMon.

We prototype CoKeMon design on Arm platforms using the latest hardware extension [2]. We demonstrate three kernel monitors that showcase the use of CoKeMon: log auditing, dynamic kernel integrity, SMMU monitoring. We showcase that it is easy to prototype monitor logic, bare-metal, or on top of a monitor runtime of the developer's choice (e.g., microkernel [7]) to gain further privilege separation within the monitor service. Enabling CoKeMon incurs minimal changes to the Linux Kernel (~500 LoC). We evaluate the performance of CoKeMon on an Arm platform [48] and outline the feasibility on RISC-V. We observe no performance impact on the Linux kernel when all monitoring is disabled. For our three monitors, we report overheads ranging from 0% to 13% for realistic applications and workloads. Further, CoKeMon does not accumulate overhead when running multiple monitors.

In summary, CoKeMon makes three main contributions:

- We present the notion of decoupling monitoring from isolation in the context of kernel monitoring.
- CoKeMon shows the feasibility of this principle by removing the management layer, while still providing features such as permission views and device support.
- We prototype CoKeMon on Arm and showcase three representative monitors on real-world workloads, with minimal code changes to Linux. CoKeMon is open source at https://cokemon-sectrs.github.io.

2 Kernel Monitoring

Modern monolithic operating systems support an extensive range of functionalities, necessitating millions of lines of code. This large code base inevitably leads to an increase in the number of security vulnerabilities. To preserve the integrity of operating systems against such threats, kernel developers have implemented hardening techniques to mitigate entire vulnerability classes [19]. However, they still leave opportunities for exploitation by malicious actors that can exploit such security vulnerabilities (CVE-2024-50066 [35], CVE-2019-19241 [36], CVE-2025-6018 [30]). To address these concerns, a second line of defense is to monitor the kernel during its execution, to either detect or prevent malicious behavior.

2.1 Need for Isolation

Common objectives of kernel monitoring include auditing the system execution to detect any anomalies, validating critical operations to intercept potential exploits, or protecting privileged operations, each in the form of a monitoring service. Next, we present kernel monitoring objectives.

Auditing. A common practice is to deploy an auditing service (e.g., Auditd or eBPF) to filter and collect system events for Linux. These methods record execution traces of the Linux activity, enabling experts to analyze them later and gain insight into the system behavior. This approach is particularly useful for diagnosing the root causes of system corruption and preventing it in the future. However, using Linux subsystems makes these solutions fragile against kernel-level privilege escalation. Attackers can tamper with the monitoring module and remove traces of their intrusion. Consequently, we must run monitoring modules in their own isolation domain, isolated from the kernel they audit. Moreover, the trace logs must reside on storage that the kernel cannot access.

Dynamic Kernel Integrity. A bug in the kernel opens the possibility for an attacker to escalate their privilege at runtime. To address this, the monitoring module validates critical kernel operations, such as modifying system registers, and executes them on the kernel's behalf. Further, the monitoring must isolate itself from the kernel to stop the attackers from bypassing the validation.

Challenges. All the previous examples have one thing in common when they consider a kernel-level attacker: the monitoring module must run in isolation to be safe against a malicious kernel. Additionally, in configurations where multiple monitoring modules are running on the same system, they should ideally not have to trust each other, and all run in their own isolation domain. First, this introduces a challenge of creating multiple and mutually distrusting isolation domains. This also entails ensuring a secure switch between a service and the kernel, as well as across services. Second, each monitoring module requires a concept of a memory view on the kernel to perform its duty. This memory view must

encompass all the data structures that the monitoring modules necessitate to either audit the kernel or validate its operations. Depending on the use-case, the memory view must also contain the Memory Mapped I/O (MMIO) region associated with a device, to control this device.

2.2 Examples of Monitors

We detail existing monitoring services from prior works.

Auditing with a kernel-level attacker. There are several kernel auditing solutions in the literature [3, 25, 37, 56]. An auditing framework generates log messages that the kernel writes to a memory region visible to the monitoring module. The kernel then triggers a switch into the monitoring module's isolation domain. Once in that domain, the module starts executing and stores these messages in a secure storage region inaccessible to the kernel. In this arrangement, we consider the kernel trustworthy and reports accurate information to the monitoring module until it is compromised. Once an adversary reaches kernel-level privileges on the kernel, they may attempt to conceal their actions by altering or erasing the logs, which are indicative of a breach. However, as the attacker cannot tamper with previously stored logs, an analyst can diagnose the traces related to the malicious activities up to the point of compromise, enabling post facto breach detection.

Dynamic Kernel Integrity. Prior works have proposed integrity monitoring of the kernel at runtime. A common approach consists of preventing the kernel from modifying its own code by marking code related pages as read-only in the page tables [14, 15, 18, 20, 54]. Additionally, all writable regions of the kernel memory must not be executable, ensuring only the original code of the kernel can run. All modifications to the kernel code must go through the monitoring module, which verifies their validity against predefined policies. A critical component of this technique is preventing the kernel from altering the page table permissions. To that end, one can mark the memory regions containing the kernel's page tables as read-only, forcing every modification to be validated by the monitoring module. Finally, the implementation removes the kernel authority over the MMU by hooking into all operations that manipulate it. These hooks redirect the control flow to the monitoring module, which performs these operations on behalf of the kernel, only after verifying compliance with the restrictions described above. For example, when the kernel updates the page table base register, the monitoring module has to verify that the new page table does not contain a writable entry to the kernel's code or to any executable regions of the kernel memory.

Table 1: Comparison of different monitoring mechanisms

Name	Permissions	Management	Boundaries	Switch
TZ-RKP [14]	Coarse	Firmware	Static	Architectural
SKEE [15]	Coarse	Nested Kernel	Static	Software
Nested Kernel [20]	Coarse	Nested Kernel	Static	Software
EKC [54]	Coarse	Nested Kernel	Dynamic	Software
Veil [12]	Fine	Hypervisor	Dynamic	Architectural
Hilps [18]	Coarse	Nested Kernel	Static	Software
xMP [46]	Fine	Hypervisor	Dynamic	Architectural
Tide [57]	Coarse	Nested Kernel	Dynamic	Software
CoKeMon	Fine	None	Static	Architectural

2.3 Goals for CoKeMon

We make two observations about the monitoring module from the above examples: (i) it must run in its own isolation domain; and (ii) it needs a memory view of all the structures it has to manage. Thus, when both modules are enabled, they should be isolated from each other, and each of them should have a module-specific memory view. We aim to address these challenges in CoKeMon. In the rest of the paper, we will refer to the monolithic OS kernel as *kernel*. We will refer to each of the monitoring services, such as auditing or dynamic kernel integrity, as *M-module*.

We aim to build CoKeMon such that it isolates the M-modules from a potentially compromised kernel. CoKeMon allows deployment of multiple M-modules that are further isolated from each other. In doing so, CoKeMon adheres to the Principle of Least Privilege (PoLP), i.e., each module can only access a subset of memory that is strictly necessary for its own operations. Further, we do not want to introduce a highly privileged software component that is in charge of orchestration, management, and/or deployment of all the M-modules. Lastly, CoKeMon should use existing hardware features without incurring hardware changes.

2.4 Threat Model & Assumptions

We consider a software attacker who can exploit vulnerabilities in the kernel. We assume that the M-module provides its monitoring services correctly. If a M-module has a bug, CoKeMon cannot ensure the correctness or availability of the services offered by this M-module. However, it ensures that a buggy M-module cannot attack other M-modules. We trust the hardware, including devices, to be bug free and implemented correctly, and we consider side channel and physical attacks as out of scope.

3 Observations

In this section, we discuss the requirements for designing CoKeMon. We provide key observations from decisions made by previous works that offer insights for CoKeMon design (Table 1).

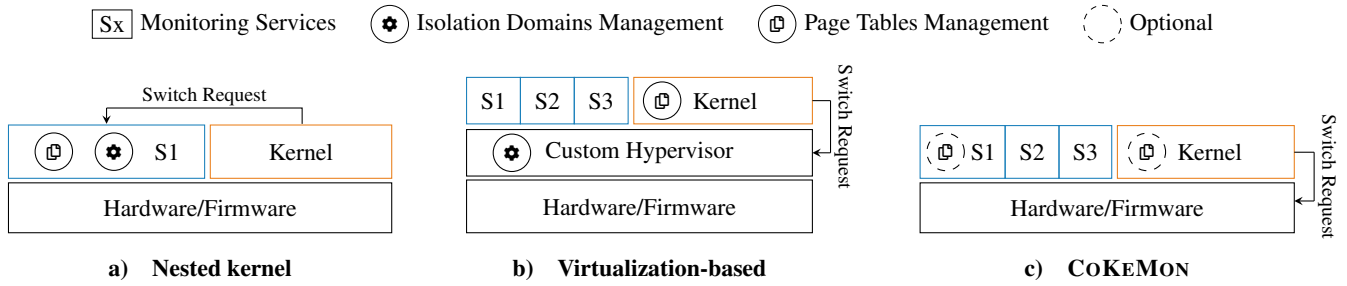


Figure 2: CoKeMon compared to Nested Kernel and Virtualization-based methods.

3.1 Permissioned View for M-modules

We have established that the M-module must be isolated from the kernel. However, the M-module may need to access the kernel memory to perform its tasks. One approach is to configure the M-module to access the entire kernel memory, but such over-allocation of permissions contradicts the Principle of Least Privilege and can turn a single M-module bug into a system-wide vulnerability, even if the M-module does not interact with critical kernel structures. To limit the impact of potential bugs in the M-module, one should tailor each M-module's permissions for the specific operations it conducts.

Consider the example of auditing, where the kernel writes the logs in the shared memory region and the M-module reads the logs to subsequently store them in a persistent storage device, out of the reach of the kernel. Thus, the auditing M-module and kernel simply need a shared region. Other examples that fall under the same category include M-modules for cryptographic operations, where the keys are kept by the M-module and the kernel sends inputs to the M-module; network filtering, where the M-module controls the network interface controller (NIC) and transmits the packets to the kernel. Thus, several M-modules simply need a small shared memory view of the kernel.

Consider the example of dynamic kernel integrity, where the M-module needs access to particular parts of the kernel memory (e.g., page tables). Other examples that fall under this category include M-modules to check the values/integrity of certain objects (e.g., kernel code) periodically or after critical events. We can limit this view to a part of the kernel, depending on the kernel structures the M-module has to read or write to perform its tasks. Lastly, there might be cases where the M-module may need access to the entire kernel memory (e.g., VM introspection).

These examples illustrate that M-modules have different consequences if compromised. A security vulnerability in an auditing M-module causes this specific service to fail, without causing a system wide compromise. In contrast, a bug in a dynamic kernel integrity M-module represents a major failure point, as an attacker can modify critical kernel sections. For this reason, we want to clearly separate the permission views of the different M-modules.

Observation 1. Different monitoring services have different permission requirements

3.2 Monitoring vs Isolation

There are two main approaches for kernel monitoring. Both of them tightly couple the monitoring aspects (e.g., access to kernel memory) with enforcing isolation between the kernel and the M-module.

Virtualization. Virtualization-based methods use a privileged component, typically a hypervisor, to isolate the M-module from the kernel. In addition to isolating the M-module from the kernel, the hypervisor has to carry out other tasks (e.g., running device drivers [56], managing M-modules lifecycle [12, 27, 46]). This expands the interface size between the kernel and the hypervisor, thereby increasing the risk of malicious actors exploiting potential vulnerabilities in the management logic and/or the M-modules to break the security enforcement. We illustrate the Virtualization-based approach in Figure 2(b).

Nested kernel. Nested kernel removes the capacity of the kernel to manage its memory translation table. It forces the kernel to go through an inner kernel when it needs to update its page tables [14, 15, 18, 20, 54]. The inner kernel isolates itself from the kernel by enforcing that no page table mappings map to its memory space, effectively enforcing its own protection from the kernel. To achieve this, the inner kernel must also instrument the kernel binary to make it trap on all instructions that can modify the MMU configuration. Additionally, marking the kernel code as read-only prevents an attacker from inserting page table instructions to bypass the inner kernel. To prevent such insertions, the kernel must also disallow all other memory regions from executing privileged code (eg., PXN bit on Arm). These design choices force the inner kernel, i.e., the M-module, to be strictly more privileged than the kernel and tightly couple the isolation with the monitoring. As a result, a vulnerability in one M-module risks

compromising the whole system, including the kernel. We show the Nested kernel approach in Figure 2(a).

Observation 2. Kernel monitoring techniques introduce a higher privileged entity to monitor the system and manage isolation domains.

3.3 Static versus Dynamic Isolation Boundary

All Virtualization-based and some Nested kernel methods offer to dynamically manage isolation domains, updating the isolation boundaries during the system runtime [54, 57]. This approach brings many advantages, such as allocating secure memory on demand [12, 27], or avoiding copy overhead [56]. However, it increases the Trusted Computing Base (TCB), giving attackers more ways to compromise the system. In contrast, most Nested kernel techniques do not enable dynamic isolation but use static boundaries. While this entails fixing the set of enabled M-modules during boot, they eliminate the need for any runtime management.

Observation 3. Enforcing dynamic isolation boundaries requires runtime management, increasing the TCB.

3.4 Careful Switching

Switching between M-modules or a M-module and the kernel requires care. A switch comprises saving and restoring register states, which can encompass several instructions. Executing such instructions must be atomic from the point of view of the attacker, since letting an attacker interrupt the switching process could break the isolation. Additionally, one must guarantee control-flow integrity during the switch and only allow selected entry and exit gates. Secure switching is a central challenge when implementing the nested kernel designs. This is much less of an issue for virtualization-based monitoring, as many architectures offer different privilege modes for virtualization. Thus, an adversary with lower privilege can be stopped from interrupting the save-restore operations, which guarantees atomic switching. Reusing existing context switching mechanisms is simpler and less prone to errors.

Observation 4. Switching between isolation boundaries is challenging. We should use existing architecture mechanisms if possible.

3.5 Device Support

Some monitoring tasks may require the use of peripherals (e.g., typically a storage device for logging system events). Prior works either attach devices to privileged system components [26, 27, 56] or do not consider device support for M-modules [12, 14, 15, 18, 20, 46, 54, 57]. The difficulty resides in that the devices access system memory via DMA, bypassing the MMU. Further, the kernel configures the access control for DMA. Thus, if we want to support device attachment per M-module, we have to enforce selective memory views via DMA as well.

Observation 5. Device support in M-modules requires DMA-aware isolation.

3.6 Observations Summary

Given our goals we set out for CoKeMon in Section 2.3 and guided by these observations in Section 3.1–3.5, we adopt the following design choices: First, the isolation mechanism provides a fine-grained memory permission model, such that we can grant each M-module only the minimal permissions they require to operate. Second, we statically define the isolation boundaries. Since the M-modules are not dynamically launched/terminated and for each M-module we know its memory footprints in advance, we can establish these boundaries at boot time. We consider this decision of enforcing static boundaries an acceptable trade-off, as avoiding runtime boundary management significantly reduces the attack surface. Third, to attach devices to M-modules, we have to prevent DMA from breaking CoKeMon's memory isolation. Fourth, for switching, we rely on an architectural switch rather than a complex software mechanism to ensure that an attacker cannot interrupt the transition. Finally, given the fine-grained permissions and static boundaries, CoKeMon requires no high-privilege orchestrator to manage isolation boundaries.

4 CoKeMon Overview

We give an overview of CoKeMon design and requirements, as summarized in Figure 2(c).

If the M-module is in charge of the isolation, it can trivially increase its own privilege, contradicting Obs. 1. To address this, we decouple the isolation mechanism from the monitoring. In other words, each M-module only embodies the code required to do its job. If a M-module needs access to (parts of) the kernel for its functionality, such a selective view is pre-configured during the launch of the M-module. This way, we achieve the said decoupling.

If we isolate the M-modules and the kernel by placing a software component beneath them, we essentially replicate virtualization-based solutions, and their drawbacks we outline in Section 3.2. To address this, our insight is to use a purely hardware-based isolation mechanism without any software layer that has to perform the isolation management (e.g., configuring page tables when switching between M-module and

kernel). Instead, our firmware only performs a context switch, which entails save-restore of registers and static switch of isolation boundaries—both of which are anyway part of a typical firmware codebase. Further, this choice also allows us to piggyback on the atomicity guarantees of the firmware. Lastly, we carefully design the interfaces and switch logic to ensure that interrupts and entry/exit code cannot bypass the atomic switch.

Hardware Requirements. We desire a hardware-based isolation mechanism that can partition the physical address space into at least two mutually exclusive regions, one for the kernel and one for the M-module. Further, the hardware should allow the creation of a shared memory region, such that both the M-module and kernel can access it. Optionally, support for permissioned memory views (read-only vs read-write) over such a region can be useful for performance. More importantly, the hardware should ensure that the M-module and the kernel do not have the privileges to change the ownership or boundaries of the physical address space.

Supporting Devices in M-modules In CoKeMon, we enforce that the M-module maintains exclusive access to devices it uses. This is a conscious choice that is necessary for security in CoKeMon, otherwise a kernel can use shared devices to attack M-modules. For the same reason, apart from the devices it owns, the M-module treats all other devices as untrusted. To enforce this ownership model for devices, a M-module must exclusively own the MMIO region of each device it controls. Additionally, it may only issue DMA to the memory it owns. Symmetrically, other M-modules and the kernel must be prevented from using DMA to access the M-module's memory.

CoKeMon Invariants. Based on our insights, we define the six invariants that a design must satisfy to meet CoKeMon goals:

INV$_{priv}$: No privileged component performs management or runtime monitoring duties.

INV$_{view}$: Each M-module has a static and selective view of the kernel.

INV$_{gate}$: The kernel and M-module have fixed and well-defined entry and exit gates; the only allowed control transfers are to the start of a gate.

INV$_{com}$: An interface between the M-module and the kernel can only pass values via registers or shared memory, which are subjected to checks in the M-module before use.

INV$_{atom}$: From the point of view of the M-module and the kernel, the switch is atomic.

INV$_{dev}$: A M-module must own an entire device if it wants to use it.

CoKeMon Guarantees. CoKeMon isolate the kernels and the M-modules, while allowing M-modules to have a selective view of the kernel. At boot, it applies these views from a user-supplied configuration file that also lists the hashes of the kernel and the M-modules. Beyond isolation, CoKeMon provides the mechanism to switch between the kernel to a M-module, and allows M-modules to own a device. Users implement the M-modules, as well as the sizing of their memory. We provide reference M-module implementations in Section 7. These M-modules implement features such as Dynamic Kernel Integrity or Auditing, as described in Section 2.2.

5 CCA-based design

In this section, we will describe how we adapt the Arm Confidential Computing Architecture (CCA) to concretize the design of CoKeMon.

5.1 CCA Background

Arm CCA is an extension of the Armv9-A architecture that introduces the notion of the Realm and Root security states. Those two states come in addition to the pre-existing Secure and Non-Secure states in Armv8. In Arm CCA, the firmware assigns a memory region to a physical address space (PAS). A core may access a PAS only when its state permits it. For example, a core in Root state can reach a PAS in any state, while a core in Non-secure state can only access Non-secure PAS. On top of the four states previously mentioned, the firmware can assign the Non-accessible state to a PAS. The Non-accessible state means that no core can access this PAS, independently of its state.

To track the security state of each PAS, the firmware encodes the states in the Granule Protection Table (GPT). Each physical core can have a dedicated, per-core GPT. The GPT is then used by the Granule Protection Check (GPC). The GPC checks upon every memory access if the core trying to perform the operation is running in the correct state.

The switch from one state to another, less privileged, state uses the Secure Monitor Calls (SMCs). A caller issues a dedicated SMC instruction to trigger a context switch to the firmware in the Root state. Additionally, the caller passes arguments in the general purpose registers, which the firmware uses to decide which actions to take. Upon reception of the SMC, the firmware saves the caller context and performs some tasks depending on the SMC arguments. Once the firmware completes those tasks, it either restores the context of the caller or the context of another security state. When restoring the context of a different security state, the firmware might forward some arguments passed by the original SMC caller.

SMMU. The System Memory Management Unit (SMMU) is an IOMMU in the Arm paradigm. It translates device spe-

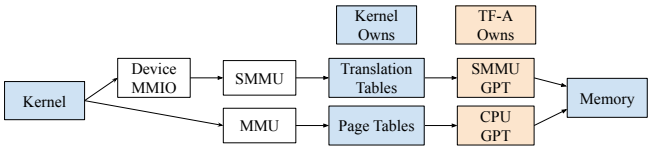


Figure 3: Permission checks in Arm CCA when a CPU issues a memory access. For CPU/device accesses, the MMU/SMMU first checks permissions using the page/translation tables. Then the MMU/SMMU enforces the GPT permissions. The memory access completes only if both checks succeed.

cific virtual addresses into physical addresses, similar to the MMU but focused on devices and their DMA activities. The SMMU enforces access permissions set by the kernel, ensuring devices do not access unauthorized memory spaces. Those permissions are encoded in per-device *translation tables*, an equivalent of the page tables for the MMU. The SMMU exposes its configuration registers through an MMIO region. The kernel writes to this *SMMU MMIO* region to configure the location of the translation tables or of the command queue. In Arm CCA, in contrast to the MMU, the SMMU has a single GPT that is shared across all CPU cores. This means that this GPT is independent of the GPT of the CPU cores using the SMMU. Figure 3 illustrates how Arm CCA combines SMMU/MMU and GPC checks together.

5.2 CoKeMon with CCA

Arm CCA offers four distinct isolation domains, Non-Secure, Secure, Realm, and Root, meeting our requirement for a minimum of two. However, the CCA states come with multiple permission combinations between those states. For example, the Non-Secure state, typically hosting the kernel, is strictly less privileged than all the other states. This imbalance in privileges between isolation domains fails to satisfy Invariant INV_{priv}. Additionally, the firmware updates the boundaries dynamically, conflicting with Invariant INV_{view}. Furthermore, we will run into issues when accommodating more than two partitions for multiple monitoring services (see Section 10).

Selective Privilege Isolation To address these limitations, we employ a Multi-GPT design. This design consists of creating multiple GPTs, in our case, one per M-module and one for the kernel. Each GPT encodes a different set of permissions, allowing a CPU running in a given state to access different memory regions depending on the active GPT. To change the active GPT, we issue an SMC to request it from the firmware. The firmware then enables a new table by loading its address in a dedicated register. During a GPT switch, the firmware role is minimal. It executes a fixed sequence of operations and never makes decisions about isolation domains. With this

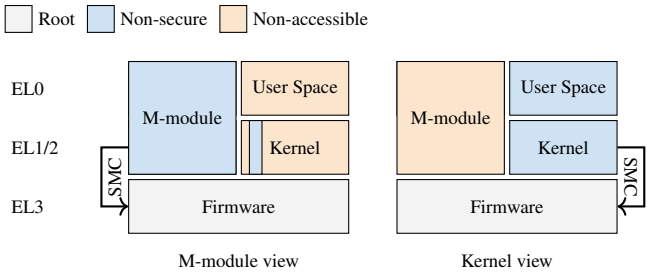


Figure 4: CoKeMon design on Arm CCA. We model the kernel and M-module views with the security states of CCA. The domains can switch from one view to the other by issuing an SMC call to the firmware.

design, no management or monitoring logic resides in the firmware. As a result, the Multi-GPT setup satisfies Invariant INV_{priv} in CoKeMon.

Furthermore, as Arm CCA isolates its different execution domains at a 4KB granularity, CoKeMon fulfills INV_{view} by selectively defining which pages of the kernel the M-module can access. We illustrate in Figure 4 how we use CCA security states described in Section 5 to model CoKeMon's selective privilege isolation.

Switching. CoKeMon requires switches between the M-modules and the kernel. To achieve this while ensuring Invariants INV_{gate} and INV_{atom}, we rely on SMCs. When the kernel or a M-module wants to transfer control, it issues an SMC instruction. Execution then jumps to the firmware and later resumes at the instruction following the SMC. This gives the M-modules and the kernel fixed entry and exit points, satisfying Invariant INV_{gate}. During the switch, the firmware changes the active GPT to the one associated with the target M-module or kernel. Interrupts are disabled while the firmware runs, so the switch cannot be preempted by another core. Thus, each switch appears atomic to both the kernel and the M-modules, satisfying Invariant INV_{atom}.

Communication. During the switch, the kernel and the M-modules communicate via the general purpose registers, as in the original SMC mechanism. For data exceeding the size of the registers, the domains instead exchange information via a shared memory region. We mark this shared memory as non-secure in both the kernel GPT and the M-module GPTs. With this approach, CoKeMon meets Invariant INV_{com}.

Role of the Firmware. The firmware operates as a privileged component. But it does not perform any management tasks (e.g., scheduling, memory allocation), it merely switches from one context to the other, following INV_{priv}. Additionally, the firmware minimizes its involvement by neither securing nor validating communications between the M-module and

the kernel, at most it simply relays it. By relying on an already hardened system component, we reduce the risk of introducing a vulnerability while implementing CoKeMon. When available, we can further shrink the TCB and use a leaner and/or verified firmware [24].

6 Device Support for M-modules

In CoKeMon, we require a M-module to have exclusive control over selected devices (INV_{dev}). To satisfy this requirement, we make device MMIO ranges inaccessible to the kernel and to the other M-modules. Concretely, similar to Section 5.2, this is as simple as marking the MMIO range as accessible in the M-module GPT, and inaccessible in the other GPTs. While we are satisfying INV_{dev}, it does not stop the M-module from using the device it controls to issue DMA operations that writes to the kernel, thus breaking INV_{view}. To address this challenge, we have to restrict DMA so that a device can only access the memory of the device owner, i.e., the kernel or a M-module.

6.1 Design Choices

To restrict DMA, we use the SMMU. However, the SMMU uses a single GPT shared by all cores, independent of the cores' current GPT. This means that if the kernel or a M-module asks a device to perform DMA on a memory region, it will ignore their CPU GPT's restrictions. In other words, either all M-modules and the kernel can read/write DMA accessible memory, or no one can. Figure 5 illustrates this issue.

We evaluate two designs to address this limitation. In the first, we assume the M-modules own no devices. There is no need for DMA, so we mark the M-modules' memory as inaccessible in the SMMU GPT. In the second design, we introduce an SMMU M-module, to checks all modifications to the translation tables and the SMMU MMIO.

Next, we explain those two designs in detail.

Design #1: SMMU GPT. In our first design, no device is attached to the M-modules. To prevent the kernel from accessing the M-modules' memory, we mark it as inaccessible in the SMMU GPT. This is in addition to marking this memory inaccessible in the kernel GPT, as we explain in Section 5.2. This way, the kernel can still use devices to perform DMA on its own memory regions. However, if the kernel instructs a device to initiate DMA over a M-module's memory, the SMMU GPT will block it. Since there are no devices attached to the M-modules, there will never be a legitimate reason to access the M-modules' memory via DMA.

Design #2: SMMU Monitor. To allow M-modules to use devices and DMA, we introduce the SMMU M-module. We require the use of this additional M-module, as the SMMU GPT is insufficient to enforce Invariant INV_{view}. Using the SMMU GPT forces us to either allow both the kernel and the M-modules to access a memory region, or none. Therefore, we rely on the SMMU M-module to check if a device is allowed to access a memory region via DMA.

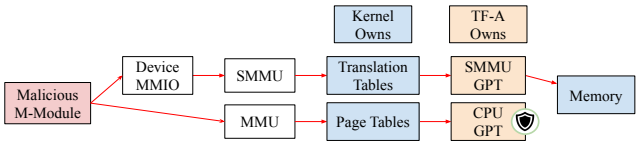


Figure 5: If a malicious M-module issues a memory request through a device, the SMMU checks its own GPT and allows the malicious M-module to access the memory, even if the CPU GPT would have blocked the same request.

During a device DMA, CoKeMon must determine if the entity that owns the device is allowed to perform this access. The best vantage point is the SMMU translation tables. Recall that the kernel controls the SMMU translation tables, and that the SMMU uses them for permission verifications before the GPC (Section 5.1). This means that if a M-module controls the translation tables, it can enforce checks on which memory devices access via DMA. We achieve this by revoking the kernel's control over the translation tables and SMMU MMIO.

First, we mark these two structures' memory regions as inaccessible in the kernel's GPT. Second, we introduce the SMMU M-module, which is in charge of managing the translation tables and SMMU MMIOs. For the SMMU M-module to perform its duty, we mark these two regions as accessible in its GPT. Figure 6 shows the GPT configurations for the SMMU M-module, as well as for other configurations. To create new SMMU mappings, the kernel must invoke a new CoKeMon API. The kernel passes the information about the mappings to the SMMU M-module. Then, the M-module ensures the validity of the information before installing the new entry in the translation table. Once the mapping is in place, if the kernel tries to access the M-modules' memory, the SMMU will not allow it, even if the SMMU GPT would allow such access.

In summary, the SMMU GPT allows access to both the kernel and the M-modules memory, and the SMMU translation tables enforce device specific memory accesses.

6.2 Optimizations

The two designs we presented in Section 6.1 have some disadvantages. The first design limits DMA usage, while the second design incurs a performance cost. We discuss two special cases where we improve our design to enable efficient DMA in the M-modules.

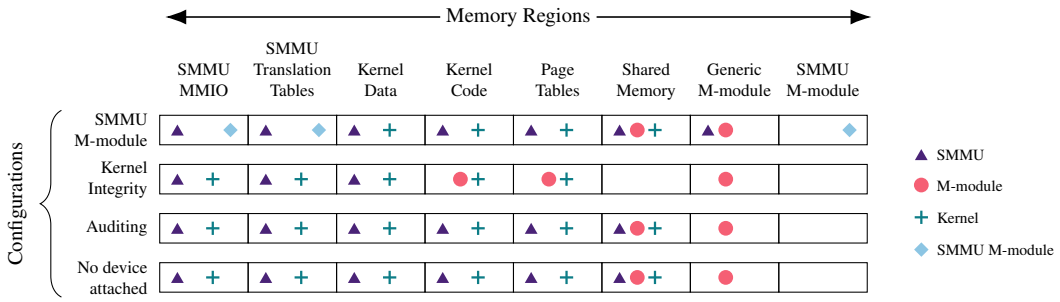


Figure 6: Access permissions across memory regions for four potential configurations. Symbols indicate which component (kernel, M-module, SMMU, SMMU M-module) can access each memory region. We encode these permissions in each component's GPT. We split the kernel memory into three regions: Kernel Data, Kernel Code, and Page Table. The last two are only relevant for the Kernel Integrity design.

Dynamic Kernel Integrity. Consider the case of dynamic kernel integrity. Here, a M-module marks the kernel code and page tables as read-only. The M-module handles all modifications of the page tables on behalf of the kernel. If we have devices attached to the kernel, we have to ensure that they cannot tamper with the kernel code and page tables via DMA. We achieve this by marking these two regions as inaccessible in the SMMU GPT. This has no functionality impact, as the kernel has no legitimate use of DMA on those two structures.

Optimization 1. When using a dynamic kernel integrity M-module, CoKeMon does not need a SMMU M-module, design #1: SMMU GPT is sufficient.

Auditing. Consider the case of auditing. The M-module stores logs generated by the kernel in a persistent storage device (e.g., SSD). The security property the auditing M-module wants to enforce is that the kernel does not erase previous logs. However, the M-module does not produce logs by itself, and accepts any logs the kernel sends. In this situation, the M-module initiates DMA to store the logs on the device. Therefore, the only DMA region the M-module requires contains the logs produced by the kernel. Since the kernel anyway produces the data, it does not matter if the kernel can access this region via DMA.

Based on this observation, we mark the logs' memory regions as accessible in the SMMU GPT. This way, the M-module can use DMA to transfer the logs to the device. Even if the kernel maps M-module's memory outside of the log region, the SMMU GPT will block these accesses. Most importantly, the kernel cannot access the storage device MMIOs and erase previous logs. Note that this optimization is specific to logging. It may not generalize for all M-modules that need access to devices.

Optimization 2. When using an auditing M-module, CoKeMon does not need an SMMU M-module, design #1: SMMU GPT is sufficient.

6.3 Interrupts

Another consideration we must address concerns device interrupts. The M-module and the kernel share the same Generic Interrupt Controller (GIC), which is common for all devices. We have to ensure that the kernel's GIC configuration does not break the functionality of the devices owned by the M-module. To solve this issue, we make a simple observation. In the context of devices, interrupts let the kernel run other processes while it waits for a device to respond. However, the M-module has no process to schedule while waiting for a device to answer. The M-module is responsible for a single task and then yields the execution back to the kernel. Therefore, when the M-module runs, we can disable the GIC, effectively masking all interrupts.

The kernel receives the interrupts on behalf of the M-module. Upon receiving an interrupt, the kernel triggers a context switch to the M-module via an SMC, informing it of the interrupt, which the M-module can then handle. By receiving interrupts addressed to the M-module, the kernel learns that a device has data to share but lacks access to the device's MMIO region, thereby remaining unaware of the actual data content. Conversely, if the kernel switches to the M-module by faking an interrupt, the M-module checks the MMIO region, finds no data, and returns control to the kernel. If the kernel receiving interrupts from a device owned by a M-module is not desirable, the M-module can poll the device's MMIO region instead of relying on interrupts.

7 Implementation

We describe our CoKeMon prototype implementation.

Prototype Platforms. We build our prototype of CoKeMon across two platforms: the Armv-A Base RevC AEM Fixed Virtual Platform (FVP) to evaluate functional correctness, and the RK3588 Radxa Rock 5B for performance measurements. We select the Rock 5B to take advantage of OpenCCA [17], a framework designed to estimate the performance of prototypes based on Arm CCA. We use Linux v6.12 as per the OpenCCA build. OpenCCA also requires a modified version of Arm Trusted Firmware-A (TF-A), which serves as the firmware.

Building Blocks for CoKeMon. To implement our switch mechanism, we change the firmware to add a new Secure Monitor Call (SMC) that triggers a switch from one domain to another. In addition to the usual context saving done by the firmware, we also save EL1 registers. We reuse the existing Multi-GPT implementation that comes with OpenCCA. We then encode the view of the kernel by the M-modules in additional GPTs, one for each M-module. Since the GPTs are not functional on OpenCCA, and for simplicity, we did not implement a fine-grained GPT for the M-module, and just marked the whole kernel memory as accessible to the M-module. A functional implementation would define fixed memory ranges where the kernel would install the structures it shares with the M-module.

Building Blocks for M-modules. The M-module developer is free to choose a runtime of their choice (e.g., bare-metal libraries, library OS). For simplicity of development, CoKeMon offers the L4Re microkernel as a M-module runtime [7]. We port L4Re to the RK3588 SoC, using the BSP porting guide. First, the firmware verifies the hash of the M-modules. Then, the L4Re kernel, Fiasco, initiates its usual boot sequence before launching the service responsible for switching to Linux. Note that, at this step, L4Re can optionally measure the hash of the Linux binary to verify its integrity. After completing the necessary preparatory steps, L4Re triggers a switch to the firmware, which starts the Linux boot process. Linux then boots normally, potentially collaborating with L4Re if the use-case requires it.

Configuring CoKeMon. To configure CoKeMon across reboots, we need a configuration file for the firmware. This file indicates the GPT layout, the active M-modules, and hashes for both the M-modules and the kernel. The firmware uses it to set up the system at boot time. To securely update the configuration, the firmware contacts a remote trusted party or uses an external, encrypted file bound to a hardware-stored key. As this is a one time cost that occurs only when one wishes to update the configuration, and it does not impact the runtime performance, we did not implement this configuration file in our prototype.

Next, we describe the different M-modules we implement for our CoKeMon prototype on OpenCCA. Most of our modifications to L4Re or Linux are specific to the M-module.

M-module #1: Dynamic Kernel Integrity. We implement dynamic kernel integrity as described in Section 2.2: we mark the Linux page tables and code read-only and hook on all system register changes that impact the MMU. For that, the M-module boots first and hashes the Linux binary. If the hash matches known values, the M-module assumes the kernel will cooperate and start the Linux boot sequence. Once Linux finishes booting, it marks all of its page tables as read-only and transfers the execution back to the M-module. Then, the M-module scans the Linux binary and replaces all MMU related instructions with hooks. We implement the hooks with SMC instructions that trigger a switch to the M-module when executed. The SMC instructions encode information about the instruction they replaced in their immediate value. In particular, we encode the type of the instruction, as well as its operands. When compiling Linux from our prototype, all MMU instruction operands are general purpose registers from *x0* to *x4*. This means that the firmware can forward their value to the M-module without further modifications. Upon a switch, the M-module validates the instruction parameters and executes the instruction on behalf of Linux. Since the firmware saves and restores the registers, the M-module writes to data structures that the firmware will use to restore Linux registers.

We note that we need to configure the GPTs so the M-module can access the kernel page tables. This also means that we have to estimate the size of memory required for page tables, over-provision the memory, and then cap the number of processes. Finally, as per prior work [15, 20, 54], to support dynamic loading of Loadable Kernel Modules (LKMs), the M-module needs to scan the new code for the presence of MMU-related instructions. Additionally, one needs to overcommit some memory at boot reserved for loading LKMs, which the M-module will have access to. We do not implement this specific functionality in CoKeMon, and leave it up to the M-module developer.

M-module #2: Auditing. We implement a M-module that safeguards logs produced by Linux, as described in Section 2.2. As L4Re does not provide storage drivers for the Rock5B board, we use another device driver instead. We send all messages immediately when the logs are written to the shared buffer, without batching.

M-module #3: Device Protection. We implement the SMMU M-module, as described in Section 6.1, to evaluate its overhead. We identify two locations in the Linux kernel that change SMMU mappings. For each location, we invoke the SMMU M-module to check if the memory region that Linux is trying to map is within the range of the M-modules.

8 Security Analysis

We reason about four attack vectors, namely M-modules attacking each other, an attacker exploiting a bug in a M-module, a compromised kernel attacking the M-modules, and attacks specific to our use-cases.

8.1 Isolating M-modules

A M-module may try to access the memory of another M-module. CoKeMon uses GPCs to enforce isolation between the M-modules and prevents them from directly accessing each other's memory. Next, a M-module may attempt to tamper with the GPTs that the GPCs rely on to decide which accesses to allow. Only the firmware can change GPTs, and in CoKeMon, we do not provide any interface from the M-modules to the firmware that allows such changes.

The M-modules may try to trick the firmware into modifying the GPTs. CoKeMon offers a limited firmware interface to the M-modules, and none of the calls use data from the M-modules. The firmware performs a save-restore of registers when switching contexts between kernel and M-modules, but the firmware never uses any of these registers. Lastly, the M-modules cannot update any EL3 system registers.

A M-module can try to interrupt the firmware in the middle of a GPT switch. This is not possible as interrupts are disabled when executing in EL3. Likewise, a M-module cannot jump in the middle of the switch sequence, CoKeMon only allows control-flow via clearly defined entry and exit gates.

Finally, a M-module owning a device can use DMA to access the memory of another M-module. This is not possible as we either block the access via the translation tables controlled by the SMMU M-module, or via the SMMU GPT, depending on the use-case.

8.2 Bugs in the M-modules

We consider the impact of potential bugs in a M-module (e.g., memory/type safety violation, bad logic) and how a kernel-level adversary can try to exploit such a bug.

First, the M-modules sanitize all inputs from the kernel, making it difficult for a kernel adversary to get control over a buggy M-module. Despite this, if a kernel attacker still succeeds in exploiting a bug in the M-module, it can only compromise the service of the buggy M-module. All other M-modules are safe, since they are isolated from each other using the GPCs (Section 8.1). For example, exploitation of a bug in the logging M-module does not affect the network filtering or dynamic kernel integrity M-modules. However, successfully exploiting bugs in certain M-modules can be fatal. For example, if the attacker can corrupt a M-module that controls the page table, then it can corrupt and control the entire kernel.

Thus, the impact of a buggy M-module depends on the operations it performs and the memory view it has. By isolating the M-modules and limiting their memory views, CoKeMon largely reduces the impact of buggy M-modules. For M-modules that have a view of critical structures (e.g., page tables), since they are small in TCB and light-weight, they can be coded with memory and type safety enforcements as well as subject to formal verification; thus reducing or even removing the probability of buggy code.

8.3 Kernel Adversary

We analyze the consequences of a compromised kernel. To protect the M-module memory from a compromised kernel, we isolate the M-modules and the kernel with the GPCs. The kernel may attempt to reconfigure the GPTs, but only the firmware can change their configuration. Moreover, CoKeMon does not provide any interface between the kernel and the firmware that can be used to modify the GPTs. The kernel may try to manipulate the firmware to change the GPTs. In CoKeMon, the firmware does not process any inputs coming from the kernel. Similarly to Section 8.1, the kernel cannot interrupt the GPT switch, since the interrupts are disabled in EL3, and cannot jump in the middle of a switch sequence, since we have defined entry and exit gates.

A compromised kernel may try to access M-module's memory by maliciously programming benign devices to perform DMA. CoKeMon subjects device-side accesses, including DMA, either to GPC or translation tables checks via the SMMU. If we are relying on the translation tables, we use the SMMU M-module to prevent the kernel from adding malicious entries. Therefore, only the devices owned by a M-module can access the M-module memory.

8.4 M-module Specific Adversaries

We consider M-module specific attacks.

Auditing. A compromised kernel may try to overwrite previous logs or to rollback the storage device state. CoKeMon prevents this since the M-module controls the storage device and does not allow the kernel to perform any accesses, which are necessary for overwriting or rollback. The M-module owns the device MMIO region, which is protected by the GPCs. The M-module is programmed carefully to ensure that when it writes new logs to the device, it does not erase previous ones. Lastly, the kernel has no interfaces it can use to trick the M-module into tampering with the logs on the device.

Dynamic Kernel Integrity. When booting the kernel, a modified kernel may attempt to set up writeable page tables, which during execution can subvert the page table monitoring. However, in CoKeMon, the M-module ensures that the kernel image abides by the requirement of setting page tables

as read-only and only then launches the kernel. This ensures that the page tables are initialized as read-only.

Next, at runtime, a compromised kernel can try to execute instructions that corrupt or bypass page tables (e.g., change page table base, add corrupt mappings, double map kernel pages in user space, or turn off the MMU). CoKeMon disallows the kernel from executing any such instructions. Further, we prevent any DMA operations from writing to read-only pages or page tables by marking them as inaccessible in the SMMU GPT (Section 6.1). Then, all page table operations go through the M-module, which in turn checks if the requested changes are benign and only then proceeds to perform them.

Lastly, the kernel may try to bypass the M-module validations for page table operations. One way to do this is to jump to existing instructions in the kernel code. CoKeMon removes all such unsafe instructions from the kernel code before booting, thus making such attacks impossible. Then the kernel may try to introduce these instructions during runtime (e.g., by writing to its code pages or jumping into the middle of existing instructions that are decoded to disallowed instructions). CoKeMon also defends against such attacks. The M-module controls the page tables; it makes all code pages read-only and data pages non-executable. This blocks modification of the kernel code. As for jumping in the middle of an instruction, this is not possible in Arm64 as all instructions are aligned and have a fixed length.

9 Evaluation

In our evaluation, we use both microbenchmarks and macrobenchmarks to assess the performance impact of CoKeMon. We run our experiments on the Rock5B [48] with 4 Arm Cortex-A76 at 2.4 GHz and 4 Arm Cortex-A55 at 1.8 GHz, 16 GB of RAM, and a 2.5 Gb Ethernet port. The microbenchmark analysis focuses on quantifying the overhead associated with a single context switch. In contrast, the macrobenchmarks examine the impact of CoKeMon on representative workloads. Moreover, we measure the boot time of CoKeMon, including the initialization of both Linux and L4Re.

Trusted Computing Base Modifications. For both the TF-A and Linux, we count the line changes with respect to OpenCCA as the baseline. For CoKeMon, we change a total of ~500 LoC in Linux, two thirds of those changes are specific to use-cases. We change ~300 LoC in the TF-A. Concerning the L4Re microkernel, we modify ~100 LoC in Fiasco, mostly to make it compatible with the Rock5b compatible with Fiasco. Additionally, we implement both a user space and a kernel space service for each M-module: ~350 LoC for dynamic kernel integrity, ~80 LoC for auditing, and ~100 LoC for the SMMU M-module. We report results for the kernel space services, as they perform better.

Table 2: Detailed cost of a round-trip to a M-module.

Routine	Cycles
Save EL1 System Registers	336
Restore EL1 System Registers	288
Save EL2 System Registers	698
Restore EL2 System Registers	657
Save/Restore General Purpose Registers	1128
GPT Switches	102
TLB Flushes	1919
Full Round-Trip Switch	**5128**

Table 3: Boot time overhead of L4Re (L+L), hash verification (Hash), and dynamic kernel integrity (DKI) compared to the unmodified OpenCCA Linux (Linux).

	Linux	L+L	Hash	DKI
Time (s)	8.36	8.97	10.77	14.47
Overhead	-	7.3%	28.8%	73.1%

9.1 Lifecycle Breakdown

Boot. We evaluate the overhead of booting the L4Re Microkernel before Linux. We report the average over 10 boot sequences in Table 3. For the baseline, we take the boot time of the unmodified OpenCCA Linux. Next, we record the time when L4Re runs first. Then, we add the cost of L4Re hashing the Linux binary. Finally, we measure the boot time of L4Re configured for Dynamic Kernel Integrity.

Kernel Switch. We evaluate the cost of a single round-trip to the M-module, without performing any work in the M-module. We measure the average for 1 million round-trips across 10 runs. This gives an average cost of 5128 cycles per complete round-trip. Additionally, Table 2 details the cost of each step in the switch operation for a single round-trip. We observe that the cost is split more or less equally between saving and restoring the states and flushing the TLB.

Relative Kernel Switch Cost. We compare the cost for a round-trip in CoKeMon with prior work (Table 4). Nested Kernel designs avoid using architectural switches and instead rely on software gates, which are much faster (typically a sub-thousand cycles), as they do not require privilege changes. The hypervisor-based approaches (e.g., Veil [12]) have expensive context switches, since they incur a privilege level change which includes register save-restore and TLB flushes. We see a similar cost in CoKeMon. TZ-RKP, which is the closest in this regard to CoKeMon, does not perform a TLB flush and is thus faster. Similarly, the cost of such a TLB

Table 4: Cost of a round-trip compared to prior work

Name	Cycles	Switch	Freq	Arch
TZ-RKP [14]	2000	Firmware	2.30	Arm32
SKEE [15]	813	Nested Kernel	2.70	Arm64
Nested Kernel [20]	473	Nested Kernel	3.40	Intel
EKC [54]	14040	Nested Kernel	1.80	Arm64
EKC [54]	7800	Nested Kernel	1.00	RISC-V
EKC [54]	3120	Nested Kernel	0.40	RISC-V
Veil [12]	7135	Hypervisor	3.00	AMD
Hilps [18]	823	Nested Kernel	1.15	Arm64
xMP [46]	18000	Hypervisor	3.60	Intel
Tide [57]	177	Nested Kernel	2.60	Arm64
CoKeMon	5128	Firmware	1.80	Arm64

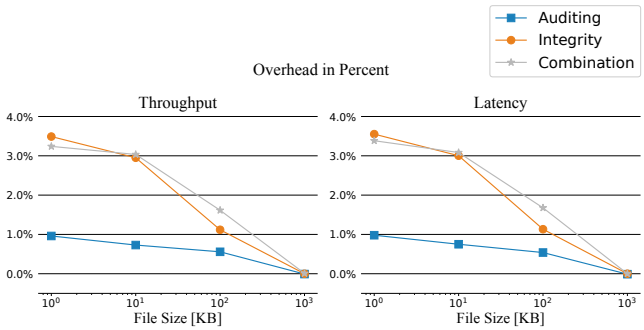


Figure 7: Throughput (left) and latency (right) overhead of Apache when running with an auditing and a dynamic kernel-integrity M-module.

flush is also noticeable in EKC [54], which is a nested kernel design. We note that Arm CCA adds a new instruction to invalidate GPT entries in the TLB for a physical address range (TLBI RPALOS) [8]. Since OpenCCA does not emulate this instruction, we have to invalidate the entire TLB. Consequently, we expect physical range-based invalidation to improve CoKeMon performance.

9.2 M-module #1: Dynamic Kernel Integrity

We evaluate the overhead of the dynamic kernel integrity use-case, with SMMU optimizations from Section 6.2. The dynamic kernel integrity M-module includes going through L4Re to update the page tables and to modify MMU related system registers. Among the system register instructions we replaced in Linux, we find *TTBR0* and *TTBR1*, which Linux updates on every context switch. This leads to a noticeable slowdown in microbenchmarks, which we quantify in the next paragraph. However, later we also show that real-world workloads amortize this cost.

Context Switch Overhead. We evaluate the impact of validating *TTBRX* changes on every context switch. We observe an overhead of 1372% when running 2 processes, which reduces to 505% when running 16 processes. We expect such

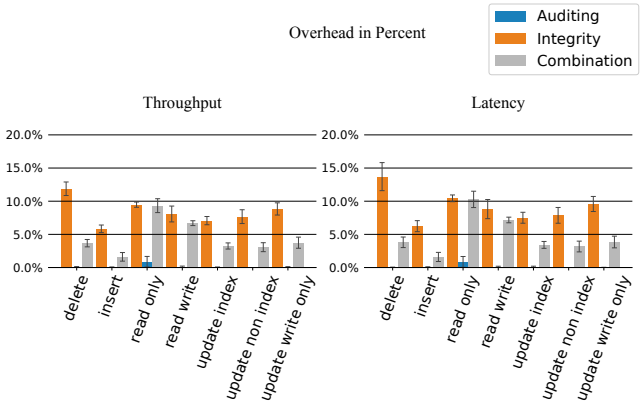


Figure 8: Throughput (left) and latency (right) overhead of PostgreSQL when running with an auditing and a dynamic kernel-integrity M-module.

costs, as we replace a single register write with two security state switches.

Creating & Executing a New Process. We measure the cost of creating a new process. We spawn `hello-world` and `/bin/sh` binaries with `fork+execve` from LMBench [43]. We report an overhead of 116% for `hello-world` and 92% for `/bin/sh`, which captures the cost incurred by the M-module for validating the new page tables as well as updating the VA-PA mappings as the child process executes. The cost of CoKeMon checks has a larger impact when spawning smaller processes, as expected.

Apache. We run an Apache web server version 2.4, and measure its throughput and latency with ApacheBench [1], using 10000 requests over a 1 Gbps network and file sizes from 1 KB to 1 MB. Apache runs unmodified and with its default configuration. We report the overhead of CoKeMon compared to the OpenCCA baseline in Figure 7. For small files, the cost is up to 3.5% for both metrics, but it drops as file size grows and is negligible beyond 1 MB. The reason is that, for large transfers, network processing dominates, while the number of M-module switches only depends on the number of requests, so the switching overhead is amortized.

PostgreSQL. We evaluate the throughput and latency of a PostgreSQL server version 17.6 for different types of local SQL transactions using sysbench [13]. We execute each type of transaction 6 times for 5 minutes on a database with 10 tables of size 100000 each. Similar to Apache, PostgreSQL runs unmodified and with its default configuration. As shown in Figure 8, the overhead for latency and throughput is about 9%, with throughput ranging from 5.8% to 11.9% and latency from 6.2% to 13.0%.

9.3 M-module #2: Auditing

We evaluate the impact of protecting logs produced by Linux. Similar to the previous M-module, we use the SMMU optimization from Section 6.2. As a baseline, we send all the logs to the device directly. For CoKeMon, we send the logs to a kernel module, which initiates a switch to the M-module. The M-module then writes the log messages to the device. For both case studies, we run our benchmarks with the same configuration as listed in Section 9.2.

Apache. We record every request the server receives with Apache's built-in logging system. We present the results in Figure 7. We observe an overhead of 1% for throughput and latency, both going to 0% for files larger than 1 MB. As with dynamic kernel integrity, for large transfers, the network processing cost dominates.

PostgreSQL. We record the logs using the built-in logging collector. We configure it to record all data definition and data modification statements. We present the results in Figure 8. The overhead is 0% for most benchmarks, as the cost of switching is negligible relative to processing log messages. The *read_only* benchmark is an exception, as it does not produce logs and therefore has a lower overall execution time. With this lower baseline cost, variations between runs become more pronounced, and the measured overhead remains within the margin of error of our measurements.

9.4 Combining M-module #1 and #2

To assess the impact of running two M-modules, we simultaneously enable both modules. Each M-module has its own GPT and runs on top of a separate L4Re kernel. We report the results in Figure 7-8. For PostgreSQL, the combined M-modules' cost is essentially that of the auditing M-module, with an overhead of at most 10.3%. PostgreSQL generates many log messages, and auditing costs more than dynamic kernel integrity, so it accounts for most of the combined cost, which drives the overhead down. There is one outlier for *read_only*, as this benchmark does not produce any logs, the combined cost is simply equal to the cost of the dynamic kernel integrity. Apache emits fewer logs compared to the number of context switches. Consequently, dynamic kernel integrity is the primary cost driver, with overhead up to 3.4%.

9.5 M-module #3: Device Protection

We evaluate the overhead of using the SMMU M-module. As a baseline, we perform the same operations, but without the SMMU M-module. We measure the time to write files ranging from sizes 1MB to 1GB. We use an NVMe disk as the storage device that is connected to the SMMU on the Rock5B. We report the results in Table 5. The overhead is influenced by the file size. This is not surprising as growing file size increases the number of new SMMU mappings that need to be added, where each addition incurs a call to the M-module. We notice a larger overhead for 10MB. On further investigation, we find that the device driver optimizes the mappings. Thus, the overhead is proportional to the number of new mappings.

Table 5: Overhead of using the SMMU M-module to write to an NVMe.

File Size [MB]	**1**	**10**	**100**	**1000**
Overhead	1.80%	7.49%	3.49%	6.15%

10 Related Work & Discussion

Kernel monitoring has been studied mainly in two aspects: mechanisms to enable monitoring and developing the monitoring services. Hypervisor-based VM introspection is popular due to its simplicity, where a privileged hypervisor can monitor the virtual machine execution, including the guest kernel [34,50,51]. Further, as summarized in Section 3.2, monitoring mechanisms can use virtualization [12,46] or Nested Kernel [15,18,20,54]. In CoKeMon, we de-privilege the M-module and remove the need for management. CoKeMon implement representative monitoring M-modules; they can be adapted to implement improved designs [25,45,56].

There are several approaches to reduce the impact of kernel bugs. On Intel, BULKHEAD [29] isolates components of the kernel with the Protection Keys for Supervisor-mod (PKS). Both on Arm and RISC-V, the CHERI architecture [52] introduces bound checks in the hardware for all memory operations, which enables the creation of compartments. On Arm, HAKC [42] uses the Memory Tagging Extension (MTE) and Pointer Authentication Code (PAC) to create kernel compartments. Finally, independent of the underlying architecture, LXDs [44] uses microkernels to isolate kernel subsystems. CoKeMon is not a generic compartmentalization framework for the kernel, instead it is designed specifically for monitoring. On one hand, this allows us to make design decisions such as static partitioning and selective views, which are specific to our problem setting. On the other hand, we have to reason about DMA-capable devices, which bypass the CPU.

Verifying M-modules. Our design choice of decoupling M-modules from the isolation mechanisms is inspired by prior works in other domains [21,24]. By virtue of this insight, we partition and reduce the TCB of CoKeMon. Even with a reduced TCB and lack of management, CoKeMon still assumes that the M-module code is bug free. To this end, one can employ verification techniques independently for each M-module implementation, thus allowing scalability.

Furthermore, prior formal verification efforts have shown the feasibility of verifying the Arm CCA software stack, including the firmware [40], which is also relevant for CoKeMon. In addition, it is common to employ bug detection techniques such as fuzzing and symbolic execution [28, 38, 39].

Limitations. CoKeMon makes deliberate trade-offs that introduce some limitations. First, we fix the set of M-modules at boot, and changing them requires a reboot. Second, we reserve each M-module's memory at boot and do not allow it to grow at runtime. This static setup removes runtime management, which reduces the TCB and the attack surface, but it forces users to fix resources in advance. In practice, M-modules have small, predictable memory requirements, which makes our choice reasonable. Lastly, new use cases require new M-modules. While we provide several reference implementations to build on, a user will have to implement their own M-module for new monitoring use-cases.

Alternative Hardware Enforcement for CoKeMon. Our current CoKeMon implementation uses Arm CCA. In particular, we rely on the GPCs to partition the physical memory. The GPC mechanism is a satisfactory hardware primitive for CoKeMon, as it enables a 4KB isolation granularity and imposes the same restriction on the DMA operations. However, CoKeMon design is independent of this particular hardware and can be realized on other platforms. In general, CoKeMon can use a hardware mechanism that offers: at least two isolation domains, a selective view from one domain on another, and the ability for an isolation domain to own a device.

Concretely, on RISC-V, we can implement CoKeMon using the Physical Memory Protection (PMP), which supports 8–64 isolated regions [32]. Each isolation domain will correspond to a PMP region. The M-module's selective view on the kernel can be encoded by PMP permissions [23, 55]. We can implement the kernel view by assigning to the kernel memory a region with the permissions `read-write-execute`. Two higher priority regions can hide the firmware and the M-module memory from the kernel. In the M-module view, we can have the kernel region marked as `--`, and the M-module region as `read-write-execute`. An extra PMP region can represent the view of the M-module on the kernel. As in CoKeMon, the M-module can own a device by adding its MMIO range to the M-module's PMP region and protect device DMA (e.g., with IOPMP) [16, 22, 33, 49]. Unlike Arm CCA, RISC-V PMP can only protect contiguous memory regions, which cannot tolerate fragmentation. This does not limit CoKeMon's applicability as we do not perform any dynamic memory allocation.

On Intel, we can implement CoKeMon using Intel TDX [31]. TDX isolates domains by encrypting their memory with distinct keys. To give the M-module a selective view on the kernel, the kernel's memory will need to be split across two keys: one for pages shared with the M-module and one for pages hidden from it. However, while the TDX module is open-source, we cannot test a modified module without Intel's signing key. We leave the implementation and evaluation of these alternatives to future work.

11 Conclusion

We present CoKeMon, a system that decouples kernel monitoring from isolation. To this end, we leverage existing hardware techniques and ensure that there is no privileged management layer. Our evaluation showcases the feasibility of CoKeMon on real-world applications with low overheads.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their feedback. We are grateful to Andrin Bertschi for insightful discussions on OpenCCA and for detailed comments on the paper; to Mark Kuhne for discussions on RISC-V, extensive feedback on the paper, and assistance with debugging; and to Supraja Sridhara for helpful feedback on the paper.

Ethical Considerations

In conducting and presenting this research on CoKeMon, we have carefully considered the ethical implications of our work. We present here our analysis of stakeholders, potential impacts, mitigations, and our rationale for proceeding with this research.

Stakeholders

We have identified the following stakeholders who may be affected by our research:

- **Operating System Developers and Maintainers**: Organizations and individuals who develop and maintain monolithic operating systems like Linux, who may need to understand and potentially address the vulnerabilities our work highlights.
- **Security Researchers and Practitioners**: Individuals and organizations working on kernel security who may use our techniques to develop better defensive mechanisms.
- **Cloud Service Providers**: Companies operating cloud infrastructure where kernel vulnerabilities could be exploited, and who may benefit from improved monitoring capabilities.
- **End Users**: Individuals and organizations using systems with monolithic kernels who depend on kernel security for data protection and system integrity.

Impact

Positive Impact

CoKeMon provides a novel approach to kernel monitoring that addresses limitations in existing solutions. By decoupling isolation from monitoring, we enable more flexible and less privileged monitoring solutions, reducing the attack surface of kernel security mechanisms. This advancement directly benefits system security by providing better detection and prevention of kernel exploitation. By eliminating the need for a privileged management layer, CoKeMon reduces the Trusted Computing Base, thereby decreasing the potential for security vulnerabilities. Our research contributes to the broader understanding of isolation mechanisms and monitoring architectures, potentially inspiring future research in secure system design.

Negative Impact

While our research primarily focuses on building a mechanism to make systems more secure, we understand that by publishing we leave room for misinterpretation and misimplementation of our design.

Neutral Impact

Our work requires minimal changes to existing systems, making adoption feasible without major disruption.

Mitigations

While our work does not identify specific new vulnerabilities requiring disclosure, we have been careful to discuss attack scenarios in a defensive context, focusing on how to prevent rather than execute attacks. Throughout the paper, we explicitly frame our work as a defensive mechanism. We provide detailed implementation guidance for defenders while avoiding step-by-step exploitation instructions. We explicitly acknowledge the limitations of our approach, including the requirement for hardware support and the need for static isolation boundaries, helping implementers understand when CoKeMon is and is not appropriate.

Decision

After careful consideration of the ethical implications, we decided to proceed with this research and its publication. We believe the defensive benefits of CoKeMon outweigh the potential for misuse. Given the widespread use of monolithic kernels in critical infrastructure, improving their security monitoring capabilities serves the public interest.

Open Science

The source code of CoKeMon is available at https://doi.org/10.5281/zenodo.20402220. The source code contains:

- fvp: Modification to test CoKeMon on the Armv-A Base RevC AEM Fixed Virtual Platform (FVP)):
  - linux: Linux kernel based on Arm reference solution (https://gitlab.arm.com/linux-arm/linux-cca) with modifications to switch to a M-Module
  - tf-a: Arm TrustedFirmware-A (TF-A) based on Arm reference solution (https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git/) with modifications to support the switch to a M-Module
- l4: L4Re system for both the FVP and the Rock5B:
  - fiasco: The L4Re microkernel (https://l4re.org/) with support for the Rock5B and for CoKeMon
  - l4re: L4Re userspace:
    - pkg/bootstrap: Modifications to boot on the Rock5B
    - pkg/osswitch_*: 3 CoKeMon M-Modules
- rock5b: Modifications to test CoKeMon on the Rock5B:
  - linux: Linux kernel based on OpenCCA (https://github.com/opencca) with modifications to switch to a M-Module and support for 3 CoKeMon M-Modules
  - tf-a: Arm TrustedFirmware-A (TF-A) based on OpenCCA (https://github.com/opencca) with modifications to support the switch to a M-Module

References

[1] ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4 — httpd.apache.org. [Accessed 29-10-2025]. URL: https://httpd.apache.org/docs/current/programs/ab.html.

[2] Arm Realm Management Extension (RME) System Architecture. https://developer.arm.com/documentation/den0129/latest/. [Accessed 05-02-2026].

[3] Forward secure sealing — lwn.net. [Accessed 14-11-2025]. URL: https://lwn.net/Articles/512895/.

[4] Log4j vulnerability - what everyone needs to know. https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know. [Accessed 05-02-2026].

[5] NVD - cve-2016-5195. https://nvd.nist.gov/vuln/detail/cve-2016-5195. [Accessed 05-02-2026].

[6] NVD cve-2021-44228. https://nvd.nist.gov/vuln/detail/cve-2021-44228. [Accessed 05-02-2026].

[7] The L4Re Operating System Framework. [Accessed 12-11-2025]. URL: https://l4re.org/index.html.

[8] TLBI RPALOS, TLB Range Invalidate GPT Information by PA, Last level, Outer Shareable.

[9] Understanding and mitigating the Dirty Cow Vulnerability. https://www.redhat.com/en/blog/understanding-and-mitigating-dirty-cow-vulnerability. [Accessed 05-02-2026].

[10] The Linux Kernel surpasses 40 Million lines of code: A historic nilestone in Open-Source software. https://www.stackscale.com/blog/linux-kernel-surpasses-40-million-lines-code/, 2025. [Accessed 05-02-2026].

[11] Server Operating System Market Volume, Share & Industry Analysis. https://www.fortunebusinessinsights.com/server-operating-system-market-106601, 2026. [Accessed 05-02-2026].

[12] Adil Ahmad, Botong Ou, Congyu Liu, Xiaokuan Zhang, and Pedro Fonseca. Veil: A protected services framework for confidential virtual machines. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3623278.3624763.

[13] akopytov. GitHub - akopytov/sysbench: Scriptable database and system performance benchmark — github.com. [Accessed 29-10-2025]. URL: https://github.com/akopytov/sysbench.

[14] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660267.2660350.

[15] Ahmed M. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *Network and Distributed System Security Symposium*, 2016. URL: https://api.semanticscholar.org/CorpusID:8991310.

[16] 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 Symposium (USENIX Security 21)*. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani.

[17] Andrin Bertschi and Shweta Shinde. OpenCCA: An open framework to enable arm cca research. In *8th Workshop on System Software for Trusted Execution (SysTEX 2025)*, 2025.

[18] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. Dynamic virtual address range adjustment for intra-level privilege separation on arm. In *NDSS*, 2017.

[19] Kees Cook. Kernel hardening: Ten years deep - kees cook, google. URL: https://www.youtube.com/watch?v=c_NxzSRG50g.

[20] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. *SIGARCH Comput. Archit. News*, 43(1), March 2015. doi:10.1145/2786763.2694386.

[21] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224056.224076.

[22] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, Wenbin Zheng, Siqi Zhao, and Haibo Chen. siopmp: Scalable and efficient i/o protection for tees. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3620665.3640378.

[23] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021. URL: https://www.usenix.org/conference/osdi21/presentation/feng.

[24] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132747.3132782.

[25] Varun Gandhi, Sarbartha Banerjee, Aniket Agrawal, Adil Ahmad, Sangho Lee, and Marcus Peinado. Rethinking system audit architectures for high event coverage and synchronous log availability. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/gandhi.

[26] Varun Gandhi, Sarbartha Banerjee, Aniket Agrawal, Adil Ahmad, Sangho Lee, and Marcus Peinado. Rethinking system audit architectures for high event coverage and synchronous log availability. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/gandhi.

[27] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. Hecate: Lifting and shifting on-premises workloads to an untrusted cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560592.

[28] Google. Syzkaller - kernel fuzzer. URL: https://github.com/google/syzkaller.

[29] Yinggang Guo, Zicheng Wang, Weiheng Bai, Qingkai Zeng, and Kangjie Lu. Bulkhead: secure, scalable, and efficient kernel compartmentalization with pks. *arXiv preprint arXiv:2409.09606*, 2024.

[30] Red Hat. cve-2025-6018. URL: https://access.redhat.com/security/cve/cve-2025-6018/.

[31] Intel Corporation. *Intel® Trust Domain Extensions*, 6 2025.

[32] RISC-V International. The risc-v instruction set manual: Volume ii. URL: https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf.

[33] RISC-V International. Risc-v iopmp specification. URL: https://github.com/riscv-non-isa/iopmp-spec.

[34] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. Sok: Introspections on trust and the semantic gap. In *2014 IEEE Symposium on Security and Privacy*, 2014. doi:10.1109/SP.2014.45.

[35] jannh. Project zero. URL: https://project-zero.issues.chromium.org/issues/371047675/.

[36] jannh. Project zero. URL: https://project-zero.issues.chromium.org/issues/42451072/.

[37] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. Sgx-log: Securing system logs with sgx. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3052973.3053034.

[38] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[39] Anil Kurmus, Andrea Mambretti, Alessandro Sorniotti, Vincent Lenders, Damian Pfammatter, and Bernhard Tellenbach. {SoK}: Automating kernel vulnerability discovery and exploit generation. In *19th USENIX WOOT Conference on Offensive Technologies (WOOT 25)*, 2025.

[40] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, Gareth Stockwell, Mark Knight, and Charles Garcia-Tobin. Enabling realms with the arm confidential compute architecture. 2023.

[41] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560585.

[42] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakcs. In *NDSS*, 2022.

[43] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*. USENIX Association, 1996.

[44] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA, July 2019. USENIX Association. URL: https://www.usenix.org/conference/atc19/presentation/narayanan.

[45] Riccardo Paccagnella, Kevin Liao, Dave Tian, and Adam Bates. Logging to the danger zone: Race condition attacks and defenses on system audit frameworks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3417862.

[46] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xmp: Selective memory protection for kernel and user space. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020. doi:10.1109/SP40000.2020.00041.

[47] Jenny Guanni Qu. Kernel bugs hide for 2 years on average. Some hide for 20. https://pebblebed.com/blog/kernel-bugs, 2026. [Accessed 05-02-2026].

[48] Radxa. Radxa rock 5 model b. URL: https://radxa.com/products/rock5/5b.

[49] 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), Nov. 2021. URL: https://tches.iacr.org/index.php/TCHES/article/view/9309, doi:10.46586/tches.v2022.i1.630-656.

[50] Fabian Schwarz and Christian Rossow. 00SEVen – re-enabling virtual machine forensics: Introspecting confidential VMs using privileged in-VM agents. In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, August 2024. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/schwarz.

[51] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. *SIGOPS Oper. Syst. Rev.*, 41(6), October 2007. doi:10.1145/1323293.1294294.

[52] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, 2015. doi:10.1109/SP.2015.9.

[53] Willie. Most Popular Linux Distributions Market Share 2026. https://commandlinux.com/statistics/most-popular-linux-distributions-market-share/, 2025. [Accessed 05-02-2026].

[54] Jiaqin Yan, Qiujiang Chen, Shuai Zhou, Yuke Peng, Guoxing Chen, and Yinqian Zhang. Ekc: a portable and extensible kernel compartment for de-privileging commodity os. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

[55] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E. Carlson, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/yu-jason.

[56] Chuqi Zhang, Jun Zeng, Yiming Zhang, Adil Ahmad, Fengwei Zhang, Hai Jin, and Zhenkai Liang. The hitchhiker's guide to high-assurance system observability protection with efficient permission switches. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3658644.3690188.

[57] Shiyang Zhang, Chenggang Wu, Chengxuan Hou, Jinglin Lv, Yinqian Zhang, Qianyu Guo, Yuanming Lai, Mengyao Xie, Yan Kang, and Zhe Wang. Tide: An efficient kernel-level isolation execution environment on aarch64 via dynamically adjusting output address size. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3719027.3765111.