ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

In Proceedings of the 34th USENIX Security Symposium (USENIX Security'25)

DORAMI: Privilege Separating Security Monitor on RISC-V TEEs

Mark Kuhne
ETH Zurich

Stavros Volos
Azure Research, Microsoft

Shweta Shinde
ETH Zurich

Abstract

TEE implementations on RISC-V offer an enclave abstraction by introducing a trusted component called the security monitor (SM). The SM performs critical tasks such as isolating enclaves from each other as well as from the OS by using privileged ISA instructions that enforce the physical memory protection. However, the SM executes at the highest privilege layer on the platform (machine-mode) along side firmware that is not only large in size but also includes third-party vendor code specific to the platform. In this paper, we present DORAMI—a privilege separation approach that isolates the SM from the firmware thus reducing the attack surface on TEEs. DORAMI re-purposes existing ISA features to enforce its isolation and achieves its goals without large overheads.

1 Introduction

RISC-V is emerging as a promising ISA for upcoming platforms, ranging from high-performance computation [34], to sensors [47], to accelerators [46], to root-of-trust [9, 18]. Given the open nature of the RISC-V development model, there are several implementations of cores that adhere to the RISC-V standard. A RISC-V platform comprises of an SoC, peripherals, and several other board components that are customized by vendors. In such an ecosystem, maintaining compatibility becomes a challenge and poses a threat of fragmentation. To this end, RISC-V standards ratify ISA specifications that the core manufacturers can follow to ensure software and compiler compatibility. Similarly, at a platform level, the firmware acts as an interface between the operating system (e.g., Linux kernel) and the underlying platform. Specifically, RISC-V offers a programmable firmware layer called the machine-mode (M-mode) that houses low-level interfaces to the CPU as well as peripherals. The software that runs in the firmware is sourced from different vendors that provide platform components. However, this opens up several security concerns. Any code that executes in the M-mode has direct access to all memory, including the one belonging to the OS and applications. Thus any bug, intentional or accidental, in third-party modules that execute in the firmware can compromise the confidentiality and integrity.

Trusted Execution Environments (TEEs) aim to isolate application memory from privileged code executing in the OS that is prone to bugs, with an abstraction called *enclaves*. Keystone has showcased TEEs on RISC-V platforms [53]. One of the main building blocks in Keystone is to use a trusted and bug-free security monitor that executes in the M-mode and creates isolated memory regions for enclaves. Keystone caters to the versatility of the RISC-V ecosystem by assuming a standard ISA feature called physical memory protection (PMP) to achieve its goals. Since only the M-mode has privileges to change PMP configurations, Keystone and other RISC-V TEEs trust not only the security monitor but any other M-mode software to not maliciously corrupt or tamper with PMP settings. However, given the nature of RISC-V firmware, any third-party code that executes in the M-mode and is part of the firmware can circumvent TEE protection.

In this paper, we aim to address this gap in the security assumptions of RISC-V TEE platforms. Our goal is to enforce privilege separation between the two components that execute in the M-mode: (a) the security monitor which performs critical operations of managing PMP configurations using privileged instructions; (b) the firmware which performs platform-specific tasks that usually do not use PMP-specific instructions. This would allow platform vendors to use third-party modules in the firmware without the need for verifying that they do not break enclave isolation. Similarly, TEE vendors can limit their testing and verification to the security monitor without having to reason about or check the effects of the rest of the firmware components. To this end, we present DORAMI—the first system that enables privilege separation of the security monitor on standard RISC-V platforms.

Privilege separation in general is a conceptually simple security principle which is often challenging and imperfect to enforce in real-world systems. To start with, practical implementations are not modular but have several dependencies and interactions between functions that need high or

low-privileges. Thus, defining a partition boundary is challenging [38]. Even after addressing the question of which functionality should reside in high and low-privilege compartments, if the code has high co-dependence, it necessitates a rich interface between compartments, leading to subtle attacks (e.g., confused deputy) [51]. Finally, the mechanisms to enforce isolation itself can be challenging. To address this, most privilege separation approaches rely on a trusted lower-layer (e.g., kernel [37] or hypervisor [43, 61] or hardware changes [36]) to enforce isolation.

Our analysis of RISC-V firmware and security monitor shows valuable insights that are conducive to privilege separation. Since the development of security monitors is specifically for TEEs, the implementation is fairly modular and indeed well-separated from the firmware. This is also partly because frameworks like Keystone aim to be platform agnostic to the best of their ability. By virtue of this, the interface between the firmware and the security monitor is also surprisingly simple—all interactions are via small set of well-defined functions that only pass data by registers without ever needing memory de-references. The OS and the applications also interact with the firmware necessitating interface-hardening. This is a standardized interface which allows the OS to interact with any platform firmware thus preserving compatibility. This provides a good vantage point for privilege separation to protect the monitor from untrusted OSes.

The main challenge in realizing DORAMI is the privilege separation enforcement itself. Since the M-mode is already at the highest privilege, short of changing hardware [36], there is no other layer that can interpose the interactions between the firmware and the security monitor. Worse yet, since both of them execute in the M-mode, they have the privileges to always use PMP-related instructions as well as access the OS and enclave memory. One viable approach is to employ intra-mode isolation, such that DORAMI only allows the security monitor to use the PMP instructions. However, such in-situ separation entails several challenges stemming from the nature of the low-level code that is trying to enforce isolation, in the presence of an adversary that can generate code, jump in the middle of instructions, trigger asynchronous events that change control flow (e.g., interrupts), and abuse the lack of atomicity. These challenges have been demonstrated in prior works that take the intra-mode isolation approach for kernels and hypervisors [35, 44].

Our main contribution in DORAMI is to address the challenges of intra-mode isolation by re-purposing an existing PMP mechanism extension called *enhanced PMP (ePMP)* [32]. We outline four main invariants to ensure that only the security monitor can access PMP registers. Next, we make the security monitor a gateway between the OS and the firmware, ensuring secure transitions. Finally, we ensure that the security monitor and the firmware do not share any state that can result in control or data flow changes (memory and trap vectors). While seemingly simple, enforcing these invariants turns out to be challenging because the security monitor has to enable and disable memory isolation to different physical address ranges as it transitions between the OS and the firmware. Our careful design of interface interposition and compartment transition using PMP has to account for subtle attacks such as ROP chains that exploit PMP instructions, trap handlers that can break out of compartments, and use of PMP instructions to avoid lock-ins.

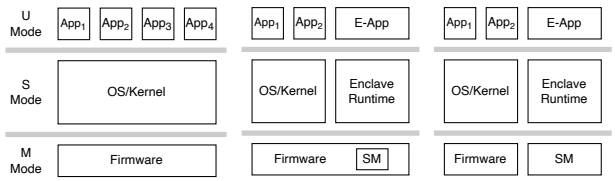
Results. We implement DORAMI using ePMP emulation on two FPGA setups (Rocket and NOEL-V cores) and QEMU. We perform end-to-end benchmarking on HiFive Unmatched board (FU740 cores) with only PMP support. DORAMI does not break assumptions of OSes (Linux kernel), applications with and without enclaves (CPU and IO, databases, web-servers), or firmware (OpenSBI).

Contributions. DORAMI is the first work that shows privilege separation of the security monitor from the firmware on a standard RISC-V platform that supports enhanced PMP. DORAMI evaluation on a RISC-V board, two FPGA-based cores, and an emulator shows that it does not incur drastic software changes or performance penalties. DORAMI is open source: https://dorami-riscv.github.io.[1]

2 Motivation

RISC-V has three execution modes as shown in Fig. 1. The machine-mode (M-mode) houses the firmware which runs at the highest privilege. It has access to all memory regions and the privilege to change machine registers (CSRs equivalent to control registers on x86_64 and special registers on Arm) [7, 14, 26]. The supervisor-mode (S-mode) has lower privileges than the M-mode and houses the OS. Applications execute in user-mode (U-mode) and the OS isolates them from each other using page tables. Since the S-mode can access any U-mode memory, an attacker that can execute its own applications can exploit bugs in the OS to bypass process isolation to compromise other applications (Fig. 1a). To reduce this attack surface, RISC-V TEEs use hardware primitives to create an *enclave* abstraction, wherein the OS executing in S-mode and untrusted applications executing in U-mode cannot access enclave memory [36, 42, 50, 53, 64]. In particular, one approach is to use physical memory protection feature (PMP)—standard feature in RISC-V privilege specification supported on most 64-bit platforms [4, 30, 31, 34]—to prohibit the OS from accessing parts of physical memory. Fig. 1b shows one such TEE based on Keystone. It introduces a trusted security monitor in M-mode who is in charge of managing PMP registers to create contiguous physical memory regions for enclaves that are inaccessible to the OS. Keystone can execute security-sensitive applications and supporting runtimes in the enclaves to achieve its security goals.

[1] Artifact is available at: https://doi.org/10.5281/zenodo.14677522.



(a) Standard RISC-V **(b)** Keystone **(c)** DORAMI

Figure 1: Software stack on RISC-V systems.

Table 1: RISC-V Firmware Analysis.

Firmware	Version	Date	Modules	Lang	Supported Platforms	LoC
OpenSBI	1.3	10.2023	8	C	All modern RISC-V boards	23,942
BBL	1.0	08.2019	0	C	Early RISC-V boards	23,060
RustSBI	0.3.2	10.2023	4	Rust	Qemu, Unmatched, Allwinner D1*, K210	–[2]
OreBoot	6b9ddbe	10.2023	8	Rust	Qemu, Unleashed, Allwinner D1*	23,552

2.1 Firmware vs. Security Monitor

On closer examination, all RISC-V TEEs rely on a security monitor—a trusted component executing in the M-mode. For example, the Keystone security monitor isolates enclaves (code and data), generates attestation reports, facilitates secure context switches between the OS and the enclave, and securely destroys enclaves before the OS can reclaim the physical memory. Other TEEs have security monitors that perform similar protections (e.g., shared memory, intra-enclave isolation, peripheral isolation). The monitors typically have small codebases (10-15 KLoC), can be programmed in memory-safe languages [1], and are subject to formal verification [55].

Other than the security monitor, the M-mode also houses the firmware. The exact functionality and codebase comprising the firmware depend on the specifics of the platform (the components on the SoC, the board, vendor-specifics, device management, interrupt controller, timer, IO, synchronization). To allow diversity of such platforms while providing a uniform interface to the OSes, RISC-V specifies a Supervisor Binary Interface (SBI) that the firmware implements and the S-mode assumes [25]. We analyzed 4 open-source firmware projects: Berkeley bootloader (BBL), OpenSBI, RustSBI, and OreBoot [8, 17, 19, 28], summarized in Tab. 1. We observe that the C implementations are capable of booting Linux, but are prone to memory vulnerabilities. While the Rust implementations aim to address this gap, they have not reached a maturity to support fully functional Linux. More importantly, the firmware comprises not only the core-functionality (e.g., firmware versioning) but also third party code specific to the components (e.g., cache controller, timer) on the RISC-V platform. For example, OpenSBI has plugins from 8 vendors. Since it executes in M-mode with the highest privileges, if there are any bugs in the firmware (accidental or intentionally introduced by third-party drivers), the attacker can exploit them to corrupt the firmware. For example, prior works have shown several bugs in firmware from devices such as IoT sensors, to phones, to accelerators [39,45,49,54,57]. The RISC-V firmware eco-system is not yet mature (e.g., no CVEs, security bulletins, advisories), so we cannot report the number of bugs in each of the implementations listed in Tab. 1. We investigated OpenSBI bugs manually by sampling the RISC-V mailing list. We found several bug reports and fixes for buffer overflows [23], missing null termination [24], NULL pointer dereference [22], missing checks [20], and incorrect masks [21]. Thus, the security monitor in M-mode can be a target of firmware exploits to bypass enclave isolation.

[2] RustSBI LoC is not precise since its build process is platform-dependent.

2.2 Challenges in Privilege Separation

Ideally, by the principle of least privilege, only the security monitor should be able to change critical information pertaining to enclave isolation. Similar to OS designs that advocate for kernel's privilege separation, such principled separation of the SM can significantly reduce not only the TCB but also the effective attack surface. To draw a further analogy to OS design, the kernel must not access user memory in order to protect itself from exploits where the attacker tricks the kernel into accessing attacker-controlled user application memory. For example, on x86_64, the kernel in ring 0 is denied from accessing ring 3 memory; instead, it has to explicitly use SMAP/SMEP [14] features to enable the access. This precaution limits the attacker's capabilities when exploiting kernel vulnerabilities. On RISC-V, the security monitor or, for that matter, any M-mode code does not need to access any S/U-mode memory unless explicitly required (e.g., when the OS requests the SM to generate an attestation measurement of an enclave via an ecall). Thus, the SM should be disallowed from accessing S/U-mode memory by default.

Partitioning the SM and the firmware into two compartments is a worthy goal if three requirements are met: the partition boundary, the interface between the partitions, and partition enforcement mechanisms. The first step is to decide which functionality of the SM and the firmware should execute with the higher privilege. From our analysis of several open-source security monitors for RISC-V TEEs [42, 50, 53, 64], we observe that the monitor typically has a clean and clear separation—it handles enclave lifecycle and accesses certain hardware interfaces, it seldom interacts with the firmware for servicing enclaves. On the other hand, the OS invokes the firmware quite often but it almost always does so for non-enclave operations. Due to this modularity, deciding the partition boundary is a relatively easy task, especially compared to similar efforts for monolithic kernels and applications [44]. The second step is to ensure that the privilege-separated compartments have a minimal interface that does not require excessive data transfers, thus further minimizing the attack surface. Our analysis shows that the SM and the firmware do not need to pass any data beyond register values, all of which are non-pointers. Tab. 2 summarizes our interface analysis for 4 open-source RISC-V TEEs.

Table 2: Number of APIs from the S-mode (OS) to security monitors (P), from P to firmware (F).

Monitor	Release	Version	Firmware	P LoC	P	F
Keystone [53]	Mar 2021	1.0	OpenSBI	8,401	10	11
Elasticlave [64]	Sep 2023	1.0	BBL	8,538	22	5
Penglai [50]	Aug 2023	Tvm	BBL	10,232	19	6
Sanctum [42]	Feb 2020	1.0	independent	5,092	31	N/A

The final step in achieving such privilege separation is enforcing the isolation. For example, one approach is to introduce a lower layer that transparently isolates two components (e.g., using a hypervisor to do intra-kernel isolation [35]). Since M-mode is the lowest level that can execute software on RISC-V, the only option is to rely on the microarchitecture which would require, perhaps undesirable, platform changes [36]. Alternatively, prior approaches achieve in-situ isolation by using hardware features [35,40,44,59,63] and instrumentation [48]. Either way, the enforcement must ensure that the attacker can never misuse the isolation-specific operations. For example, when the kernel uses SMAP/SMEP, it has code snippets that conditionally allow it to access user memory. If the kernel has memory bugs, the attacker can do a ROP-chain attack to first use the SMAP/SMEP features [13]. As another example, when the hypervisor isolates the kernel, it has to mediate all interfaces to correctly enforce the isolation. Finally, the design has to consider low-level operations such as interrupts, exceptions, and coarse-grained privileges.

2.3 Problem Statement

DORAMI aims to ensure that the monitor is privileged separated from the firmware. It introduces the notion of a *PMP compartment (P)* and a *Firmware compartment (F)*, where the PMP compartment is strictly more privileged than the Firmware compartment. PMP compartment houses the SM and does security-critical operations (e.g., enclave lifecycle) that requires access to critical hardware instructions (e.g., PMP management, trap vectors). Firmware compartment houses the remaining M-mode code i.e., platform firmware.

The OS, applications, and enclaves can invoke services from the M-mode by explicitly invoking *ecalls* to synchronously switch to M-mode. Additionally, if the CPU core receives a runtime exception (divide-by-zero) or interrupt (timer), the execution switches asynchronously to M-mode handlers. DORAMI has to ensure that any such interfaces from the S/U-mode to the M-mode are guarded. Further, DORAMI has to route the request to the correct compartment. For example DORAMI has to invoke the Firmware compartment if the S/U-mode wants to invoke firmware functions and PMP compartment if the OS wants to create a new enclave.

Since the M-mode is the most privileged layer on the platform, DORAMI has to rely on hardware for isolation. Specifically, DORAMI uses the RISC-V hardware's ability to perform fast physical memory isolation via PMP configurations. However, RISC-V allows any M-mode code to change PMP configurations. So DORAMI has to enforce intra-mode isolation to prohibit the Firmware compartment from tampering the PMP configurations. To this end, DORAMI must enforce four security invariants:

$\mathbf{I}_{\texttt{ePMP}}$: *Only P is trusted to configure and change PMP regions.* DORAMI only allows P to access PMP registers,[3] since the isolation itself is done using PMP.

$\mathbf{I}_{\texttt{EnEx}}$: *P is the only entry and exit point between S/U-mode and M-mode.* DORAMI introduces new transitions for existing interface between the S/U-mode and M-mode to ensure isolation—not only between compartments but also between the modes. This includes synchronous interfaces (ecalls) and asynchronous transitions (traps).

$\mathbf{I}_{\texttt{Intf}}$: *F can only invoke P via a fixed interface.* DORAMI provides only one interface from P to enter F, without nesting. Such explicit interface allows DORAMI to ensure a fixed entry and exit point to and from the Firmware compartment to enforce PMP isolation as well as control the data that is passed between the compartments.

$\mathbf{I}_{\texttt{MT}}$: *P and F do not share any memory regions or trap vectors.* DORAMI allocates independent physical memory regions to the compartments. It also maintains different interrupt vector tables (IVTs, referred to as trap vectors on RISC-V) per compartment. This way, if F installs malicious handlers that are triggered when P is executing, they are not executed. DORAMI performs a secure context save and restore across compartments and modes during transitions.

Threat Model & Scope. The PMP compartment is trusted by all components on the systems and is assumed to be bug-free. The PMP compartment does not trust the Firmware compartment and any other code executing in S/U-mode (OS, user apps, enclaves), and hence avoids accessing any memory belonging to these untrusted components. The Firmware compartment does not trust the S/U-mode, as is typical on RISC-V platforms. The enclaves do not trust each other or the OS and the OS does not trust the enclaves, as is typical for TEEs. The attacker has full control of the host OS, can launch malicious enclaves, and exploit bugs in the firmware with the goal of: a) executing arbitrary code in firmware; b) leak code, leak/corrupt data of enclaves. The attacker can use the Firmware compartment to compromise the PMP compartment, which is in charge of isolating the compartments and enclaves.

DORAMI does not address DoS, side-channel attacks, and hardware bugs that may break the TEE guarantees provided to enclaves. It relies on existing mechanisms that provide protection against these attacks. DORAMI does not protect against a Firmware compartment that can launch DoS attacks against the PMP compartment.

[3] Barring one case that we will explain in Section 5.2.

2.4 Existing Approaches

Tab. 3 shows prior works that enforce isolation on different architectures and layers. DORAMI shares challenges and approaches (e.g., using call gates, binary scanning, and interrupt handler for sanitization) with them. In our experience, these are common for any intra-mode isolation that does not use instrumentation [35, 59, 63]. We compare DORAMI to two approaches in particular for achieving intra-mode isolation by either modifying the hardware or by repurposing standardized hardware features. Cure [36] provides intra-mode isolation and exclusive assignment of peripheral devices to enclaves. But it modifies hardware to add CPU and bus-level checks: CPU sets enclave ID and system bus does the access control. In DORAMI, instead, our goal is to provide intra-mode isolation in the most privileged layer while relying on standardized hardware features (i.e., ePMP) and without any additional changes to hardware. Our work shows how to overcome challenges arising from this goal (discussed in Section 3).

Nested Kernel (NK) [44] isolates memory management functions in kernel space by protecting the page table translation configuration to create two compartments: nested kernel with higher privilege to perform memory reconfigurations and outer kernel that serves the remaining kernel functions. Comparing NK to DORAMI, we observe some similarities in this isolation principle, however the insights are different. First, NK operates in kernel space (Ring 0) and uses MMU and write-protect (WP) bit. This combination ensures that the attacker does not jump to the privilege nested kernel code or data. On the other hand, DORAMI operates in firmware space (M-mode) where it cannot use page-table based isolation with the MMU and there is no WP-bit; instead it has to resort to ePMP. Because of this difference, DORAMI has to address the challenge of an attacker jumping to code gadgets in a different way—instead of unmapping the pages, we reconfigure the PMP to make the gadgets inaccessible. While this may seem a minor difference, DORAMI has to introduce a PMP reconfiguration code snippet which can then in turn be used as a gadget. To make this gadget useless to an attacker, DORAMI uses a novel PMP trick—as soon as the attacker tries to misuse the gadget, it locks the attacker's memory. Next, NK reasons about interrupts by changing all the interrupt handler code to check that the WP-bit is set. This ensures that the outer kernel cannot misuse the handlers to access the nested kernel. In contrast, DORAMI allows the firmware to use its own arbitrary interrupt handlers. It simply ensures that the executing firmware can never change its own memory permissions. Specifically, DORAMI uses binary scanning, non-writeable code pages, and atomic compartment switching.

3 Rationale for using ePMP

All prior RISC-V TEEs use a standard ISA feature called physical memory protection (PMP). Although PMP can effectively isolate enclaves and the host OS from each other, it has several limitations that make it unsuitable for realizing compartmentalization within the M-mode. First, we explain the details of this PMP-based isolation and outline why it is not a good fit for DORAMI. Then we introduce another standardized RISC-V ISA feature called ePMP (stands for enhanced PMP) and explain how it can realise DORAMI.[4]

Table 3: Related Work Summary. Comparison of prior works based on hardware modifications, standardized hardware features, target ISA, privilege level for the trusted protector that enforces isolation, isolation abstractions used for the protectee, isolation enforcement type, and TCB size.

Approach	HW Mod	Std. Feature	ISA	Protector (Trust Priv.)	Protectee (Isolation Abs.)	Iso. Type	TCB KLoC
Cure [36]	✓	Bus filter	RISC-V	FW	FW, OS, Enclave	ex-situ	3.0
NK [44]	✗	MMU, WP	x86	Kernel	Kernel	in-situ	4.0
SKEE [35]	✗	MMU	ARMv7/8	Hypervisor	Kernel	in-situ	Hyp.
NeXen [59]	✗	PT nesting	x86	Hypervisor	Monitor, Domains	in-situ	N/A
Elisa [63]	✗	EPT	x86	Kernel/VM	VM	in-situ	1.4
SVA [43]	✗	LLVM	x86	SVM	OS, Services	Instr.	5.1
DORAMI	✗	ePMP	RISC-V	Monitor	Monitor, Firmware	in-situ	10.6

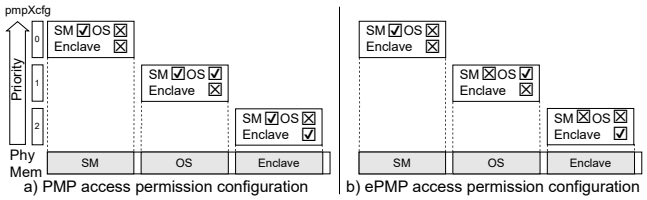


Figure 2: The system memory is divided into 3 memory regions, one each for SM, host-OS, and enclave. With **PMP**-based isolation, host-OS and enclave cannot access each others region, but the SM always has access to both, regardless of the configuration in the `pmpXcfg` registers, where `X` is a placeholder for a specific entry. With **ePMP**-based isolation, the SM can also only access the memory in its own region.

Background: Physical Memory Protection (PMP). It is a feature for RISC-V CPUs that can be used to divide the physical memory into *PMP regions*, where a region is defined by a range of continuous physical addresses. Each of these regions can then be associated with specific access permissions defined by a PMP configuration register. RISC-V introduces two new types of registers, `pmpaddr` and `pmpcfg`, to enable PMP regions and region-specific configurations, respectively. Since these are privileged registers (CSRs), only M-mode is allowed to change them. Once PMP regions are set up, on each memory access originating from S/U-mode, the hardware checks if the target address is protected by a PMP region. If so, it further checks whether the access type (R/W/X) is permitted according to the PMP configuration.

PMP can be used to isolate M-mode from S/U-mode, such that the hardware denies any attempts by unprivileged software (e.g., OS) to access M-mode memory (e.g., preventing modifications of privileged system components). Specifically,

[4] The RISC-V specification refers to the ePMP extension as Smepmp.

M-mode first sets up one PMP region to protect itself by setting the PMP configuration to be non-readable, non-writable, and non-executable, as shown in `pmp0cfg` in Fig. 2(a). Then, to allow the S/U-mode to access its own memory, M-mode sets up a second PMP region for OS and user-space, as per `pmp1cfg` in Fig. 2(a). With this PMP configuration, any access by S/U-mode to SM will cause the `pmp0cfg` check to fail, thus stopping accesses to M-mode. On the other hand, when the S/U-mode accesses region 1 (i.e., its own memory), `pmp0cfg` does not apply; instead, `pmp1cfg` applies and allows the access. In other words, by setting up regions and assigning permissions using priority-based PMP configurations, one can isolate the M-mode from S/U-mode on RISC-V.

Region configurations can overlap; in this case the hardware verifies an accessed memory address against the first matching configuration with the highest priority. Further, one can create multiple isolated regions within the S/U-mode by using multiple PMP configurations. Existing TEEs, such as Keystone, use this to create exclusive PMP regions for each enclave by flipping the access permissions between enclave and OS regions during context switch. This is required to establish memory isolation between OS and enclaves, as PMP only differentiates between M-mode and S/U-mode, but not between different entities within S/U-mode. At time t_1, before executing the OS, the SM grants S/U-mode access to the OS and denies access to the enclave. Later at time t_2, before executing the enclave, the SM flips the permissions, allowing access to the enclave region and denying access to the OS.

Problem 1: Default access permissions of M-mode. PMP-based separation is unidirectional, i.e., only M-mode software is protected from access by unprivileged software but not vice versa. This leaves a large attack surface. We consider the following example where secure enclaves are deployed on the system. PMP enforces inter-enclave isolation, but the M-mode software is buggy. A malicious enclave exploits the bug (e.g., via ecalls) to get arbitrary code execution in M-mode (e.g., ROP chain). Since the code executes in M-mode, it can corrupt M-mode memory. More importantly, the enclave can trick the M-mode into accessing any S/U-mode memory (e.g., belonging to another enclave). Such an exploit could eventually lead to leaked or compromised data of a second enclave. If PMP-enforcement blocked M-mode's access to S/U-mode, the malicious enclave limits to M-mode.

Problem 2: M-mode PMP configurations are permanent. One way to address the above outlined problem is to enforce PMP configurations on M-mode as well. In the PMP specification, this is indeed feasible, by setting an additional bit in the `pmp0cfg` register. There are two undesirable repercussions of this: (a) once set, the PMP configurations applied to M-mode cannot be modified until reboot; (b) the configurations apply to all modes. Put together, this makes it impossible to change PMP configurations when switching enclaves (e.g., make only region 1 accessible at time t_1 and only region 2 at time t_2). In the context of our example, when the enclave tricks the M-mode into accessing enclave 2's memory, the hardware will deny it. But enclave 2 itself can never access its own memory anymore. Thus, while this mechanism effectively limits M-mode's ability to access certain memory areas, it is unsuitable for enclaves, as they require frequent PMP configuration changes (e.g., switching execution between enclaves).

Enhanced PMP (ePMP). It is a recently ratified extension to PMP [32] that applies each set of PMP configurations specifically to M-mode or S/U-mode. Fig. 2(b) shows that ePMP allows setting up the memory regions where all entities have exclusive access to their own memory regions. Further, ePMP allows dynamic changes to M-mode's PMP configuration. ePMP allows the flexibility needed to achieve DORAMI goals, but is not ideal. Specifically, it has three shortcomings: (a) any M-mode code can maliciously change regions and PMP configurations, leaving a large attack surface via buggy implementation; (b) when applied, a PMP configuration is enforced for any code executing in M-mode, without fine-grained isolation within the mode; (c) even if fine-grained isolation is possible, transitioning between two partitions requires care of ePMP configuration to ensure isolation.

4 DORAMI Compartments & Interfaces

DORAMI introduces 2 compartments in M-mode: *P* and *F* isolate the SM and firmware from each other. DORAMI enforces the 4 security invariants in Section 2.3 to effectively prevent F from maliciously tampering with SM or enclaves. However, housing *P* and *F* in M-mode requires careful consideration of the compartment memory layout, interfaces, transitions and permissions. Therefore, DORAMI places each compartment in separate physical memory and ensures that F can never access any other memory outside its own compartment, satisfying $\mathbf{I}_{\texttt{MT}}$. For inter-compartment and inter-mode communication, DORAMI provides a set of secure register-based interfaces that P always checks and controls, thus satisfying $\mathbf{I}_{\texttt{EnEx}}$. To transition between compartments, DORAMI provides a secure mechanism that never unlocks both P's and F's memory spaces simultaneously, thus satisfying $\mathbf{I}_{\texttt{Intf}}$. Finally, DORAMI introduces a binary scanner. During boot it ensures that the firmware is free from any malicious code that can change the isolation configuration, thus satisfying $\mathbf{I}_{\texttt{ePMP}}$.

Memory Views. DORAMI starts with a memory layout such that code and data of the PMP compartment are in separate physical memory regions without any overlap. Code pages are configured *r-x*, and data pages are configured *rw-*. Similarly, code and data for the Firmware compartment are separated in non-overlapping physical pages with the appropriate permissions. More importantly, DORAMI ensures that the PMP compartment and Firmware compartment do not have any code and data pages overlapping each other. DORAMI achieves these layout requirements by changing the bootloader and layout of the compiled SM and firmware binaries.

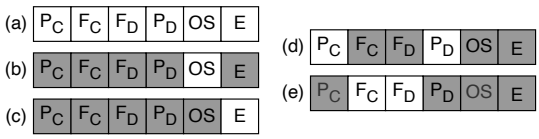


Figure 3: DORAMI access permissions when executing core. **P**=PMP compartment, **F**=Firmware compartment, **OS**=Host OS, **E**=Enclave; C=code region, D=data region; **white**=access allowed, **grey**=access denied. (a)-(c): Same memory view as in DORAMI and legacy deployment: (a): Core operates in M-mode; (b): Core executes Host-OS in S/U-mode; (c): Core executes enclave in S/U-mode. (d)-(e): Memory views added by DORAMI: (d): Core operates in M-mode and the PMP compartment executes; (e): Core operates in M-mode and the Firmware compartment executes.

During execution, DORAMI ensures an isolated memory view, as shown in Fig. 3. First, it uses default ePMP features such that when P and F are executing, they cannot access OS- and enclave memory. Next, DORAMI confines P and F within the M-mode such that each can only access their own code and data. It again uses ePMP to achieve such intra M-mode isolation but in an atypical fashion. For example, when the execution is transitioning from P to F, DORAMI disables access to P and enables access to F. More importantly, according to $\mathbf{I}_{\texttt{ePMP}}$, DORAMI only allows P to execute instructions related to PMP reconfiguration. DORAMI also preserves Keystone's isolation model, i.e., S/U-mode cannot access M-mode and enclave memory. Lastly, for every interface that involves mode or compartment change, DORAMI changes the PMP configuration to reflect the correct memory view (Tab. 4 and Fig. 3).

Interfaces. Tabs. 4-6 summarize all the interactions between the components pertaining DORAMI. As per $\mathbf{I}_{\texttt{ePMP}}$ and $\mathbf{I}_{\texttt{Intf}}$, any execution flow that incurs a memory view change has to transition through P. This allows DORAMI to ensure that P makes the required PMP updates to change the memory view without leaving any window of attack. For example, consider a scenario where an enclave wants to execute a syscall in the host OS. In Keystone, this involves transitions from enclave to SM to OS. Since DORAMI executes SM in P, a syscall interface already satisfies our requirement and is thus unaffected. Next, consider the case where the OS wants to invoke a function in the firmware. In RISC-V, the OS simply makes an ecall to M-mode where the firmware services the OS request. In DORAMI, the firmware functions reside in the Firmware compartment. If the OS directly jumps to the entry point of F, the execution will fail because the firmware pages are inaccessible. To address such cases, DORAMI enforces isolation during such transitions by first transitioning through P, satisfying $\mathbf{I}_{\texttt{EnEx}}$. This way, for our ecall, P can lock the OS memory and unlock the firmware. DORAMI remedies all such flows as summarized in Tab. 4.

Transitions between S/U and F. DORAMI has to enforce isolation between S/U- and M-mode. We observe that as per ePMP specification if the PMP region is set as accessible for a particular mode (M or S/U), the other mode cannot access it. DORAMI leverages this to its advantage. When the execution is in S/U-mode, DORAMI sets the PMP configuration such that the region covers the entire memory of the currently executing unit (either OS or enclave). Similarly, DORAMI assigns PMP regions that cover the entire memory of P's code and data. This way, when the S/U-mode switches to or from P, the ePMP enforcement automatically ensures mode-exclusive access. On RISC-V, the S/U-mode can execute an ecall to transition to M-mode. The CPU traps on the ecall instruction and uses the trap vector (stored in the `mtvec` CSR) to locate and execute the handler that corresponds to the M-mode. By $\mathbf{I}_{\texttt{EnEx}}$ and $\mathbf{I}_{\texttt{Intf}}$, DORAMI enforces a fixed entry point into P by setting the trap vector correctly so an attacker cannot tamper it. As changing the trap vector requires M-mode privileges, DORAMI ensures that only P and not F can change it.

Transitions between OS and enclave. Keystone execution flow for transitions between enclaves and OS is mediated via the SM. Since DORAMI just moves the SM into P, we do not have to change PMP switching or transition flow.

Transitions between F and P. So far we have looked at inter-mode transitions (S/U vs. M) or intra-mode transitions that are forced to incur a mode transition (OS to enclave via M-mode). However, the critical change brought about by DORAMI is the creation of intra-mode isolation between F and P. Our interface, respecting $\mathbf{I}_{\texttt{Intf}}$, only allows P to invoke F, and that only for certain services such that F performs the operations and simply returns to P. This choice reduces several attack vectors (e.g., multiple entry points to F from OS/enclave/P, or arbitrary calls from F to P). Nonetheless, DORAMI still has to facilitate a call from P into F and return from F to P. This has to be done with care for two reasons: (a) P and F execute in M-mode, and both have the privileges to change PMP configuration; (b) during transition between compartments, memory and interrupt isolation has to be enforced without leaving a window for attacks (e.g., TOCTOU, lack of atomicity).

Bootstrapping. On traditional RISC-V systems, the first stage bootloader (FSBL) loads and starts the firmware, which in turn starts the OS. DORAMI modifies this bootprocess, particularly the FSBL to enforce $\mathbf{I}_{\texttt{ePMP}}$ and $\mathbf{I}_{\texttt{MT}}$. In DORAMI firmware and SM consist of two distinct binaries. The FSBL loads both of them into memory. But before starting any software in M-mode, the FSBL scans the firmware's code blob to ensure that it does not contain any malicious gadgets that can reconfigure PMP-related registers or the trap vector. Since RISC-V instructions are 2 B aligned, DORAMI scanner checks the blob for suspicious opcode patterns in 2 B intervals. If the scan does not detect any such opcodes, the FSBL starts the SM

Table 4: Transitions between compartments, OS and enclaves, that are affected by DORAMI

Source	Destination	Event	Original Transitions	DORAMI Transitions
OS	FW	ecall	OS → FW → OS	OS → P → F → P → OS
OS	SM	Enclave create/delete	OS → SM → OS	OS → P → OS
OS	Enclave	Enclave enter	OS → SM → Encl.	OS → P → Encl
Enclave	OS	Enclave exit	Encl. → SM → OS	Encl. → P → OS
Enclave	OS	ocall	Encl. → SM → OS → SM → Encl.	Encl. → P → OS → P → Encl.
Enclave	FW	ecall	Encl. → FW → Encl.	Encl. → P → F → P → Encl.
OS	-	Any M-Trap	OS → FW → OS	OS → P → F → P → OS
Leg. App	-	Timer M-Trap	App → FW → App → OS → App	App → P → F → P → App → OS → App
Leg. App	-	Non-Timer M-Trap	App → FW → App	App → P → F → P → App
Enclave	-	Timer M-Trap	Encl. → SM → OS	Encl. → P → OS
Enclave	-	Non-Timer M-Trap	Encl. → FW → Encl.	Encl. → P → F → P → Encl.

Table 5: Unaffected transitions

Source	Dest.	Event	Unaffected Transitions
App	OS	ecall/syscall	App → OS → App
App	OS	S-Trap	App → OS → App
Encl. App	Runtime	ecall/syscall	Encl. App → RT → Encl. App
Encl. App	Runtime	S-Trap	Encl. App → RT → Encl. App

Table 6: New transitions, introduced by DORAMI

Source	Dest.	Event	New Transitions
P	F	Enter F	P → F → P
F	P	Exit F	F → P
P	-	M-Trap	impossible
F	-	M-Trap	F → F (trap handler) → F

binary.[5] The SM's initialization creates the P and F compartments by configuring the PMP registers. This creates and enforces the isolation between SM and firmware. After initializing P, DORAMI transitions to F for firmware initialization. Once F finishes, DORAMI returns to P to start the OS.

5 Inter-compartment Transitions

The transitioning process between the PMP compartment and the Firmware compartment requires differentiating between the cases of switching from P to F and from F to P to ensure a correct switching routine as required by $\mathbf{I}_{\texttt{ePMP}}$ and $\mathbf{I}_{\texttt{MT}}$.

5.1 PMP to Firmware Compartment

Switching from P to F entails four steps: removing access to P's memory regions, allowing access to F's memory regions, jumping to F's entry point, and setting the trap handler to F's view. Despite the universal trust in P, these steps require careful consideration to prevent malicious influence from S/U-mode or F.

Execution in P. P is executed either because of transition from S/U-mode or because it returned from F. When entering into P from S/U-mode, DORAMI must ensure that for every mode the permissions for S/U-mode memory regions are disabled. This PMP configuration change happens directly after entering P from S/U-mode. Thus, when P is executing, it can only access its own memory (code and data pages). Additionally, as per RISC-V ISA [26], the CPU automatically disables interrupts when entering M-mode. This means that entering M-mode from S/U-mode will automatically mask all interrupts. The M-mode has to explicitly re-enable them by writing to a specific CSR. In DORAMI, we never enable interrupts while executing in P. Thus, P never incurs traps: (a) exceptions such as divide-by-zero never arise because of careful programming; (b) interrupts such as timers, even if triggered, are masked. Lastly, P has its own view of the trap vector—it has to be set such that the ecall entry point is fixed. Put together, DORAMI ensures that P has memory and trap isolation as specified by $\mathbf{I}_{\texttt{MT}}$.

Entering F. Next, we consider the case where P invokes a function in F, which requires a transition from P to F, which entails three steps: Reconfigure PMP to deny access to P's memory regions and allow access to F's memory regions, jump to F's entry point, and set the trap handler to F's view. The order of these steps is crucial for several reasons. DORAMI does not allow F to change the trap handler to preserve security, as we will explain later in Section 5.2, so P has to change it before entering F. If P removes access permission to its own memory regions, it can no longer perform PMP configuration changes to allow access to F's memory regions. So it has to allow access to F's regions before or together with removing access to its own regions. Similarly, P cannot perform the jump to F's entry point before or after removing access to its own regions. Next, we explain how DORAMI addresses these challenges securely.

Reconfiguring ePMP. The crucial step of switching from P to F is the re-configuration of ePMP. Essentially, DORAMI has to perform a configuration transition as visualized in Figs. 3d–3e, thus stopping the execution in P and starting the execution in F. To achieve this, P needs to prepare the PMP configuration registers such that the isolation hardware enforces the combination mentioned above. P first writes required values into a general-purpose register, which it then moves into the PMP configuration register. Note that this reconfiguration can be performed in a single step to disable access to P's regions and

[5] This one-time scan at boot is sufficient because F cannot write to its own code memory or execute data memory during runtime.

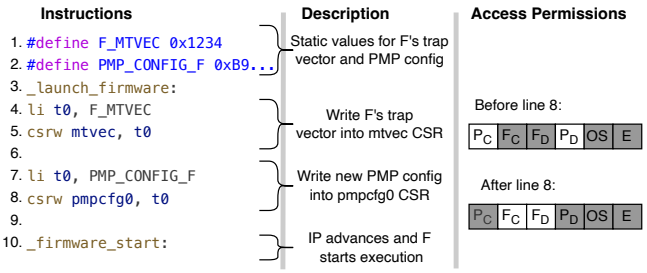


Figure 4: Transition P to F. P updates trap vector to F's view and then reconfigures access permissions for code and data regions of P and F atomically using static configuration value.

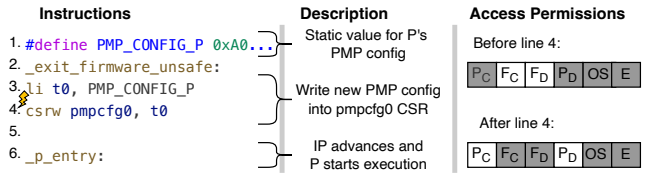


Figure 5: Unsafe Transition from F to P. F reconfigures ePMP with static value to deny access to F's regions and grant access to P's regions. ROP can exploit this transition as a gadget by jumping between lines 3-4.

enable access to F's memory regions without an intermediate step where both P and F are accessible. If DORAMI performs this step with two instructions, e.g., first allowing access to F's memory regions and afterwards removing access to P's memory regions, there would be a time window during which both regions are accessible at the same time. If now, for any reason, an exception occurs, the CPU will execute F's trap handler, as the reference to it was already set in the trap vector during the previous step. Since both P's and F's memory regions are now accessible, and F is executing, it could gain complete control over M-mode, which would contradict $\mathbf{I}_{\text{MT}}$. However, as explained above, it is impossible that DORAMI will trigger a trap (exception or interrupt) during these two operations. Hence, in this case, changing the PMP configurations does not necessarily need to be performed atomically.

Jumping to the Firmware Compartment. The last crucial step in switching from P to F is to set the instruction pointer (IP) to the start of F's code region. However, the question remains if setting the IP should be performed before or after changing the PMP configuration. The main issue is that since we set F's and revoke P's access permissions, we cannot execute code any further in P after this. Therefore, we make use of the CPU's notion of advancing the instruction pointer automatically after executing an instruction. We place the instruction that changes the PMP configuration into the last few bytes of P's code region. Executing these instructions directly at the border has the effect that we directly advance the IP from P's code region into F's code region while simultaneously re-configuring the memory access permissions. Fig. 4 shows an assembly pseudocode example of performing the overall switching from P to F.

5.2 Firmware to PMP Compartment

Switching back from F to P entails four steps: remove access to F's memory regions, allow access to P's memory regions, jump to P's entry point, set trap handler to P's view. It requires careful consideration when performing these security-sensitive tasks since the attacker controls F. The attacker cannot achieve arbitrary code injection (DORAMI enforces DEP using ePMP). But it can perform arbitrary code execution, by exploiting memory vulnerabilities to achieve ROP.

Let us first consider the step where F removes access permissions to its own memory regions and allows access to P's regions. Fig. 5 shows a pseudocode for performing this operation. First, DORAMI does not allow F to execute any instructions that change PMP configurations. Second, even if we selectively allow F to execute the particular sequence of PMP-related instructions in Fig. 5, we cannot trust F to faithfully prepare the configuration in a general-purpose register and then move it into the PMP configuration register. F can abuse this to write arbitrary values into the PMP configuration registers. Therefore, Fig. 5 provides a very expressive gadget that an attacker could use to change PMP configurations arbitrarily. This raises the dilemma that F needs to perform PMP reconfigurations, even though it is not trusted to do it.

Reconfiguration of PMP using a SallyPort The main challenge in solving this dilemma is to achieve the same effect that doing a PMP configuration change has, but with instructions that cannot be misused as a gadget. We address this by introducing the notion of a *SallyPort* region.[6] First, we construct a way to revoke F's access with a single write into the PMP configuration register that has a static value. This way, even if the attacker uses this mechanism as a gadget, it will always end up getting access to its own memory regions removed. We refer to this as a *SallyPort-Entry*, shortened to SPEntry; F can use it, but it only revokes access to regions and never explicitly adds new permissions.

Next, DORAMI has to enable access permissions of P's memory regions and jump to P's entry point. But the PMP-related instruction to enable it cannot be placed in F; it has to be executed after F is locked. We again use the insight of instruction pointer increment. We slightly modify the SPEntry such that it atomically locks F safely and unlocks P. It achieves this atomicity by putting the instructions that change PMP configurations at the end of F's code region and placing a part of P's code region immediately after the end of F (Fig. 6). This creates a layout where F is sandwiched between P's code pages. Specifically, we refer to the part of P's code region that

[6] Sally port can be entered but not exited through the same door.

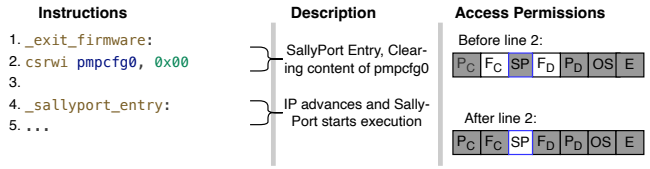


Figure 6: Transition from F to SP using SPEntry. F clears `pmpcfg0`, removing access to F's memory regions and entry that denies access to SP. Now the core can access SP.

resides immediately after F's code region as the SallyPort (SP). At this point, DORAMI can perform the remaining tasks of restoring P's trap vector, jumping to the entry point of P, and continuing its execution.

We now detail the construction of such a SPEntry. DORAMI uses a particular insight about setting PMP configurations to achieve atomic locking of F and unlocking of P. We observe that PMP bundles the permission setup for eight consecutive regions in one `pmpcfg` register. This means that memory regions 0-7 (with high priority) are configured using `pmpcfg0`, while regions 8-15 (with lower priority) are configured with `pmpcfg2`.[7] DORAMI can ensure that permissions for P and F are exclusively configured with the `pmpcfg0` register (i.e., all code and data regions for P and F are always protected by PMP entries between 0 and 7). In such a case, if F writes zeroes to `pmpcfg0`, it will have the effect that regions 0-7 are essentially non-existent, i.e., not configured. Thus, any access to memory previously covered by regions configured with `pmpcfg0` is denied if no configurations exist for them in `pmpcfg2`. We can clear `pmpcfg0` using the immediate-CSR-write instruction, which does not take an argument from a register but rather from one that is embedded in the instruction itself. Concretely, for our case, this is the instruction `csrwi pmpcfg0, 0`. This instruction cannot be used as a modifiable gadget, as it will always perform the same operation since no general-purpose register can be specified. Further, after executing the instruction that writes into the `pmpcfg` register, the CPU will increment the IP, say to an address *M*. If M falls within the range covered by `pmpaddr0-7`, the instruction execution will cause an access fault due to PMP violation. On the other hand, if M falls within the range covered by `pmpaddr8-15`, the CPU will enforce the PMP configuration as defined in `pmpcfg2`. If memory access permissions for F are completely covered with `pmpcfg0` and at least some parts of P's memory access permissions are covered with `pmpcfg2`, the above gadget will achieve our desired goal. To this end, we place the SPEntry as the last instruction in F at the end of the page followed immediately by SP. Fig. 6 showcases a full transition in pseudocode from F to SP.

Switching from the SP to P. F uses the SPEntry to switch into SP, which then unlocks P and restarts the execution of P.

[7] `pmpcfg1` is invalid / does not exist on 64-bit RISC-V CPUs

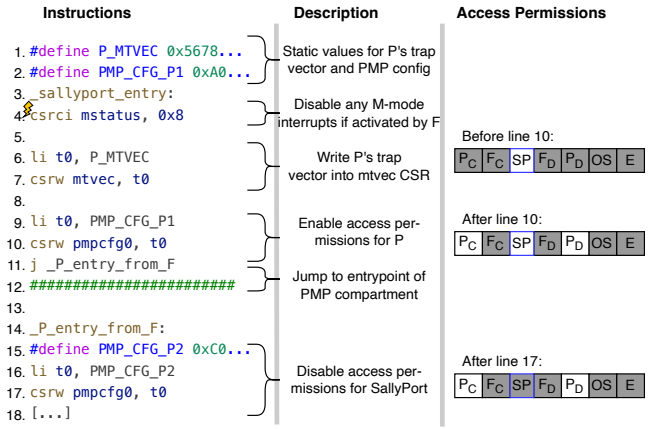


Figure 7: Transition from SP to P. The SP reconfigures the trap vector for P and restores access permissions to P's regions. Afterwards, SP jumps to P, which in turn removes access permissions for SP's memory region. An interrupt set by F could be triggered between lines 4-5.

DORAMI performs several steps in the SP. First, it disables any interrupts that might be active in M-mode. Then, it resets the trap vector to P's handler. Eventually, it reinstalls required PMP configurations in `pmpcfg0`. Finally, it jumps to a prespecified entry point address of P. Note that in this case, we are not using any tricks to advance the instruction pointer. Instead, we trust the SP to perform the PMP configuration changes faithfully and to normally jump to P's entry point. Fig. 7 shows the transition from SP to P.

Why is F not allowed to change the trap vector? Recall that F and P need to have their own views of the trap vectors for isolation. One way to achieve this is to simply save and restore the trap vector between compartment switches. However, DORAMI chooses a design where: At initialization, the trap vector is set to P's value of choice. Right before P transitions to F, it updates the trap vector to F's view that P deems safe—all handlers in the trap vector must land in F's code pages. In other words, DORAMI never allows F to change the trap vector to the extent that it employs a binary scanner to ensure that F is launched if and only if it has no such instructions. This measure is to stop the following attack.

Recall the mechanism of switching from F to P, where we use two instructions in SP to a) set the PMP configurations and b) disable any M-mode interrupts. This opens up a window where the CPU can receive interrupts (e.g., a timer activated intentionally by F just before exiting). At this point, the trap vector is still set to a value controlled by F, which could point to any arbitrary code available in SP. If the timer fires between the PMP-configuration and interrupt-disabling instructions, the CPU will execute the code pointed to by the trap vector, set by F, which will execute instructions in SP at a location of F's choice and with register states provided by F. However, if

the trap handler still points to the original location in F, as set by P before transitioning, and an interrupt arrives, the CPU will stall unrecoverably until reset, thereby preventing this attack. That is because the trap handler itself is located at an address that is now inaccessible as the PMP configuration was just cleared. So the CPU will endlessly trigger access fault exceptions. This is why DORAMI's design never allows F to set the trap vector. However, this does not prohibit F from handling interrupts and exceptions—P, trusted to set the trap vector, sets it just before entering F (as shown in Section 5.1).

Binary Scanning. We observe that, on RISC-V, the only way to modify the trap vector (`mtvec`) is to perform a CSR-write instruction for this register. Therefore, DORAMI scans F during boot to detect if CSR-write instructions are being used in combination with `mtvec`. This scan is in addition to the one that checks if F performs any modifications to the PMP configuration, with the only exception of the SPEntry.

6 Security Analysis

We argue the security of DORAMI from various attackers.

6.1 Malicious S/U-Mode

DORAMI's P ensures security of host OS and enclaves using ePMP, exactly as in Keystone [53]. The host OS or enclave runtime can try to attack the M-mode (both compartments) by directly tampering with ePMP registers. The CPU will disallow this because they do not have the privileges to access ePMP registers as per RISC-V specification ($\mathbf{I}_{\texttt{ePMP}}$).

S/U-mode can make ecalls to invoke P and misconfigure ePMPs. Since we assume that P is bug-free and performs input sanitization, the S/U-mode cannot trick P into doing their bidding (e.g., change `pmpcfg0`). S/U-mode can also try to use F to divert execution into a different entity i.e., host-OS to enclave or vice versa, without P's supervision. P prevents this attack as it strictly controls that only host OS can invoke F's functions. Further, since P is the only entry and exit point of M-mode, it can ensure that F can never compromise the execution flow ($\mathbf{I}_{\texttt{EnEx}}$ and $\mathbf{I}_{\texttt{Intf}}$).

S-mode can try to invoke Firmware compartment directly, either by jumping to the physical memory address or changing the `mtvec` register for re-routing ecalls. However, directly jumping to F's code triggers an access fault exception: (i) the PMP configuration prevents any direct accesses from lower execution levels; (ii) changing the `mtvec` register will trigger an illegal instruction exception. Since both exceptions are diverted to P, it will deny any such attempts ($\mathbf{I}_{\texttt{EnEx}}$). S-mode can use F to launch attacks against P as we discuss next.

6.2 Malicious Firmware Compartment

M-mode is used by the PMP compartment and Firmware compartment. Since P is considered trusted, we focus on the attacks that F can launch against P or enclaves. First, F can attempt to access memory beyond its PMP region configuration (i.e., the memory of P, the host OS or enclaves). The CPU triggers access fault exceptions to deny access. Although these exceptions trigger the Firmware compartment's exception handler, it cannot successfully recover since the handler itself does not contain any PMP-related instructions as confirmed from the binary scanning during boot ($\mathbf{I}_{\texttt{ePMP}}$ and $\mathbf{I}_{\texttt{MT}}$).

F's attempts to orchestrate Iago attacks to trigger PMP reconfigurations in P are likewise ineffective. P does not initiate PMP reconfigurations based on F's return values, and transitions between P and F adhere to a rigid program flow without dynamic control flow changes, as enforced per $\mathbf{I}_{\texttt{Intf}}$.

F can generate malicious instructions that alter PMP configuration or collude with the OS to copy code blobs containing them. F would attempt to write these instructions to its code or data sections and then start their execution. However, both trying to write to the code section or executing data as code will trigger an access fault exception. Similar to the previous case, the exception handler of F cannot recover from this exception since it does not contain any PMP-related instructions ($\mathbf{I}_{\texttt{ePMP}}$ and $\mathbf{I}_{\texttt{MT}}$). Maliciously using code sections from the enclave or host OS to generate malicious instructions is also ineffective, as this memory is not accessible to F. Any attempts to access leads to access fault exception that F cannot recover from.

F can attempt ROP-style attacks and execute malicious instructions as gadgets. It will be unsuccessful because: a) RISC-V has a fixed-length instruction set of 2 or 4 Bytes. Any unaligned execution will cause an exception that F cannot handle for the same reasons mentioned above. b) DORAMI ensures using binary scanning during boot that no gadgets exist that the attacker could exploit to reconfigure PMP.

F can try to execute PMP instructions in the SP by exploiting the non-atomicity of interrupt masking as follows: after switching into SP but before SP masks the interrupts, F schedules a timer interrupt that diverts execution back F's handler. F then uses the instructions in the now unlocked SP as a gadget to maliciously reconfigure PMP. This attack is not successful, as F's trap handler is no longer accessible to the CPU after switching into SP ($\mathbf{I}_{\texttt{Intf}}$ and $\mathbf{I}_{\texttt{MT}}$). So, the CPU will infinitely trigger access fault exceptions from which the CPU cannot recover until a hard reset.

F can try `mtvec` writes to modify the address of the exception handler triggered prior to invoking SP. However, binary scanning during boot prevents the existence of any such instructions in F, and generating it faces the same challenges mentioned above in regard to the PMP-related instructions.

7 Implementation

Implementing DORAMI involves assessing both the hardware features available on the targeted platform and existing software components, including the bootloader and firmware.

7.1 Components and Placement

We implement DORAMI on the HiFive Unmatched board. We extend the platform's first-stage bootloader (U-Boot SPL, provided by the manufacturer) with the binary scanner. The PMP compartment in the M-mode houses Keystone's SM, extracted from the OpenSBI extension. The Firmware compartment accommodates a modified OpenSBI version, featuring adjustments for interaction with the PMP compartment. We disable OpenSBI's functions responsible for relocating itself since we assume that the memory areas are fixed and the software in M-mode is correctly placed in memory by trusted bootloaders in earlier stages. To pass the measurement during boot, we remove any instructions related to modifications of the PMP entries or the trap vector since the PMP compartment now performs these operations instead. Data exchange between PMP compartment and Firmware compartment solely happens using registers passed during invocation of the Firmware compartment (P to F), and after it yields (F to P).

7.2 Platform Considerations

DORAMI's design requires ePMP extensions to enforce the co-isolation of the PMP compartment and the Firmware compartment in M-mode. We tested our implementations on platforms that natively support ePMP: QEMU and NOEL-V [16]. Further, we modified Rocketchip to add ePMP support. During our tests, we did not detect any memory access violations.

However, none of our ePMP-enabled platforms are suitable for evaluating DORAMI. QEMU is not cycle accurate, its performance changes based on host OS. Cores running on FPGA require significant engineering to port Linux and Keystone if they do not support it (e.g., NOEL-V). For cores that do support Keystone (e.g., Rocketchip), our ePMP implementation is not optimized and exhibits large variance even for baseline execution i.e., without DORAMI changes.

Ideally, we need a production board that supports ePMP. However, at the time of writing, we could not obtain any board that supports ePMP. Therefore, we evalaute DORAMI on a platform with only PMP support i.e., the HiFive Unmatched. While the lack of ePMP prevents a direct implementation of DORAMI as per our original design, we adapted our approach to demonstrate the feasibility. We modified the PMP configurations during context switches to exclude the L-bit, effectively activating PMP configuration enforcement only for S/U-mode. We do not expect any difference in performance in terms of executed cycles by using this board compared to one that employs ePMP (see Section 8.1).

Number of ePMP entries, enclaves, and compartments. DORAMI creates two M-mode compartments: P and F.[8] This requires 6 PMP entries during execution: 1 for non-enclave S/U-mode (e.g., OS) and 5 for M-mode (2 for P's code and data, 2 for F's code and data, 1 for SallyPort). Fig. 6–7 show this in detail. Since DORAMI needs one of these entries to be the 9^{th} entry, we need at least 9 entries on the platform. Further, each enclave needs its own PMP entries. For example, Keystone requires at least 2 PMP entries per enclave for storing enclave private and host shared data, respectively [53], whereas Elasticlave needs 3 PMP entries per enclave [64]. DORAMI does not change this requirement per enclave.

[8] See Section 9.1 for extending DORAMI to multiple F compartments.

Table 7: LoC Breakdown of M-Mode software for a standard Keystone deployment and for our DORAMI prototype. For Keystone, we removed unrelated device-specific libraries from Keystone SM for fair comparison.

Component	Keystone	DORAMI
OpenSBI	**23942**	23922
DORAMI Compat Stubs	-	46
Keystone SM	**7777**	**6777**
OpenSBI Stubs	-	**2724**
DORAMI Enforcement	-	**1156**
TCB	**31751**	**10657**
Total	31751	34625

8 Evaluation

We show impact of using ePMP instead of PMP (Section 8.1). and DORAMI's performance (Section 8.2).

TCB. DORAMI in total consists of 34.6K LoC; this includes a) the PMP compartment that houses the PMP reconfiguration functions and the SM for enclave functions and b) the Firmware compartment compartment that houses OpenSBI. Tab. 7 summarizes the lines of code of DORAMI compared to a typical system setup. We provide detailed information on the impact of the monitor components and LoC breakdown.

Bootstrapping the PMP compartment. A considerable part of the implementation overhead of P is the code required for setting itself up and for basic LibC-like functions required for operation. We implemented an initialization routine similar to the one provided by OpenSBI with 343 LoC (included in DORAMI Enforcement in Tab. 7). We import various OpenSBI functions as stubs for the SM to function correctly.

Calling Firmware Services from the Host-OS. One of P's main tasks is to forward interrupts and exceptions from S/U-mode to F, as showcased in Tab. 4. The switching process involves saving the context from S/U-mode, creating and loading a context for F and starting F. After F yields, P needs to save and evaluate the context it returns. Based on this data, P modifies the previously saved context from S/U-mode and restores the context before switching to it. This functionality is part of the DORAMI Enforcement in Tab. 7 and consists of 813 LoC: 474 LoC written in C and 339 LoC in assembly.

Handling Keystone Requests. Adding the Keystone SM to P allows it to service the OS with launching and managing secure enclaves. Including Keystone's SM into P required adding library functions from original OpenSBI (2724 LoC).

Table 8: Synthesis overview for Rocketchip and NOEL-V in different configurations. SI: Single-Issue, DI: Dual-Issue. Gray lines: maximum capacity of the FPGA.

Setup	No. PMP	CLB LUT	CLB FF	F-Mux [7+8]	DSPs	BRAMs
VCU118		1182240	2364480	886680	6840	2160
ROCKET SI, PMP	0	170642	150288	4202	36	350
	15	176307	151489	4228	36	350
	%	3.31	0.79	0.62	0.00	0.00
ROCKET SI, ePMP	15	176378	151497	4264	36	350
	%	3.36	0.80	1.47	0.00	0.00
KCU105		242400	484800	181800	1920	600
NOEL-V SI, ePMP	0	128907	73389	5053	39	84.50
	15	131973	75890	5621	39	84.50
	%	2.37	3.41	11.24	0.00	0.00
NOEL-V DI, ePMP	0	144233	77179	4165	39	96.50
	15	150841	79943	5506	39	96.50
	%	4.58	3.58	32.19	0.00	0.00

8.1 Impact of ePMP

We synthesize the custom Rocketchip CPU for a Xilinx VCU118 FPGA using FireSim and the NOEL-V CPU for a Xilinx KCU105 FPGA using Gaisler GRlib, both with a target frequency of 100 MHz. Both CPUs feature two RV64GC cores with ePMP support for up to 15 entries each.[9] We synthesize one PMP-only and three ePMP-enabled setups summarized in Tab. 8, varying from 0 to 15 PMP entries. We report the impact of enabling ePMP support on area and PMP/ePMP reconfiguration on four setups. We test our setups using the CLI Linux distributions provided by FireSim and GRlib that are compatible with Rocketchip and NOEL-V. Rocketchip operates with Linux v. 6.2 and OpenSBI v. 1.2, and NOEL-V with Linux v. 6.8 and OpenSBI v. 1.4.

Area Impact. Our first goal is to measure the impact of introducing PMP to a core. We use Rocketchip to evaluate this by configuring it with 0 and 15 PMP entries.[10] As shown in Tab. 8 (rows 4 vs 5), there is a large increase in LUT and F-Mux usage, purely due to PMP logic. Next, we measure the impact of moving from PMP to ePMP. Our Rocketchip measurements with our ePMP implementation show that this incurs relatively less hardware overheads (additionally 0.04 % LUTs and 0.85 % F-Mux). This shows that while introducing PMP logic does have high impact, extending that logic to ePMP support is not invasive. Lastly, we measure the impact of ePMP for different pipelines by configuring NOEL-V in single and dual-issue modes. When we increase the entries from 0 to 15 we see that dual-issue NOEL-V requires nearly twice the amount of additional LUTs and thrice the amount of additional F-Muxes compared to single-issue. From this, we conclude that the impact of ePMP can increase with a more complex pipeline. We note that for all our experiments, for a particular configuration of the core, the number of PMP and ePMP entries do not have an impact on DSPs and BRAMs.

Execution Overheads. Our goal is to measure the effect of frequently changing ePMP configurations and how it varies for different core implementations. To evaluate this, we used three setups: Rocketchip with our ePMP implementation, NOEL-V with in-built ePMP implementation in single and dual-issue mode. We used the AES binary from RV8 benchmark [29] and performed a configuration change for every context switch incurred for scheduling. Specifically, we invoked an `ecall` which performs a write to the `pmpcfg0` register to incur a configuration change. For each setup, we ran a baseline which performs the ecall but does not change the configuration (i.e., no writes to `pmpcfg0`). We report an overhead of 5.58% for RocketChip, whereas it is only 1.87% for NOEL-V in single-issue mode. This shows that the impact of ePMP varies based on the implementation. One possible explanation why Rocketchip introduces higher overhead is that our implementation of ePMP has not been optimized whereas the NOEL-V's implementation is production-level. Next, we report an overhead of 0.54% for NOEL-V in dual-issue mode. Thus, adding ePMP checks does not slow down optimized pipelines as much as in the case of single-issue.

8.2 Performance

We measure the lifecycle cost for booting the platform, host OS, and enclave operations. Then we benchmark stress tests for CPU with RV8 and I/O with FSCQ. Lastly, we measure the performance impact on real-world applications: webservers with darkhttpd and in-memory databases with SQLite.

Hardware & Software Setup. All our evaluation numbers are presented based on 10 runs measured on the HiFive Unmatched board four SiFive U74 RV64GC cores [31]. The U74 cores operate at 1.2 GHz with 32KB instruction and data caches, with 8 PMP entries per core. While DORAMI needs at least 9 entries which would incur two writes, our board only needs one write to update the PMP entries. To capture the cost of two writes, we add an additional write instruction in our code that performs the same write twice while ensuring that the compiler does not optimize it out. For software setup, we use the SiFive Freedom-U-SDK(2022.10) [12] for building a CLI Linux distribution (with Kernel v. 5.19) as the OS and OpenSBI (v. 1.3) as the firmware. We build two setups for the measurements: baseline and DORAMI. The baseline setup consists of a standard Keystone deployment. At the time of writing, Keystone does not provide a functional/stable base configuration for Unmatched board. Therefore, we manually add Keystone's SM to the OpenSBI version of the SDK as an extension. We modify the Kernel configuration to support contiguous memory allocations as Keystone requires. Additionally, we add Keystone's kernel module using a new yocto recipe. DORAMI setup is a standard deployment of DORAMI.

[9] Our NOEL-V synthesis for 16 PMPs failed so we used 15 for all instead.

[10] We outline security considerations for integrating ePMP into RISC-V processors in Section 9.2.

Table 9: Summary of (left) Lifecycle Costs for Baseline vs. DORAMI and (right) Impact of DORAMI on RV8 when executing in U-mode and enclave.

Stage	Component	%
Init	OpenSBI	5.29
	Monitor	91.28
	M-Mode	20.81
Kernel	Boot	0.44
	Ecall	290.00
	Timer	142.00
Enclave	Creation	15.58
	Execution	0.12
	Deletion	17.17
	Ctx. Switch	18.30

Target RV8	U-mode %	Enclave %
aes	0.35	0.28
dst.	5.58	0.29
miniz	0.14	0.63
norx	0.12	0.29
prime	0.11	0.89
qsort	0.20	0.28
sha	0.17	0.29

We place the software that handles the PMP configurations and enclave management in the PMP compartment as described in Section 4. In the Firmware compartment, we place the modified OpenSBI, which performs no PMP configurations except for using the SPEntry, as described in Section 5.2. We use the same OS as in the baseline.

Measuring Lifecycle Costs. Since DORAMI effectively acts as an additional layer in the firmware and lies on the critical path for handling M-mode interrupts or exceptions, we expect it to directly impact the overall system performance. Tab. 9 (left) summarizes the overheads of DORAMI for lifecycle operations for the platform, host, and enclaves.

Platform Initialization Cost. During firmware boot, DORAMI adds a minimal overhead of 20.81,% to set up its data structures and to set up static memory access configurations of the PMP compartment, the Firmware compartment, and host-OS. This overhead is a one-time cost during system boot-up and does not repeat until a restart of the system.

Host-OS Boot Cost. During boot, the host-OS kernel performs severeal requests to the firmware's base platform handler to get information about the system or to reconfigure settings. We measured the overhead for executing this handler, including the transitions required (Tab. 4, row 1). Although we observed a slowdown of 4-10x for these ecalls (Tab. 9 left, rows 5-6), it did not cause major overheads on the OS's boot process. The reason is that during general runtime, the M-mode is mainly invoked because of timer interrupts that the OS requests. They have a frequency of 251 invocations per second (Tab. 4, rows 8-9 for the involved transitions).

Enclave Lifecycle Costs. Before performing larger-scale benchmarks, we assessed, what overheads can be expected under normal enclave execution. Tab. 9 (left), rows 8-11 summarize measured overheads for executing the helloworld enclave from Keystone. Execution overheads are caused by handling M-mode timers in firmware and context switch overheads by DORAMI's save and restore routine, including the ecall checks. The overhead for enclave creation and deletion varies depending on the size of the loaded runtime binary.

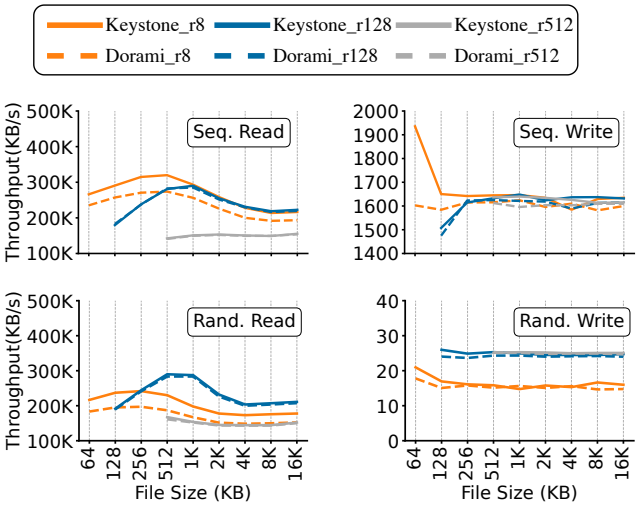


Figure 8: Throughput report for FSCQ-largefiles.

Benchmark 1: CPU (RV8). We execute the RV8 [29] benchmarks natively on the host OS and in an enclave to compare overheads for CPU-intensive applications. Tab. 9 (right) summarizes the results. For the enclaves, we additionally analyze overheads for creation, runtime, and deletion. Enclave creation adds ≈12.5% overhead, and deletion ≈10.7%. These one-time overheads result from DORAMI's direct handling of these operations in the PMP compartment, with a constant part (e.g., configuring/deleting data structures) and a variable part (e.g., checking page tables and clearing memory). For the host OS, DORAMI adds an average overhead of 0.2%, excluding Dhrystone, which incurs 5.6%. Execution overhead of other apps remains below 1% for both host OS and enclave.

Benchmark 2: IO (FSCQ-Largefiles). The RV8 benchmarks show enclave overheads induced by asynchronous context switches from scheduling. However, I/O operations require synchronous exits (ocalls) to the host OS. To measure DORAMI's impact on ocalls we use the FSCQ-largefile benchmark. It performs file I/O operations with file sizes from 8KB to 16MB and batch sizes of 8KB, 256KB, and 512KB. It reads and writes to the files both sequentially and randomly. Fig. 8 shows the throughput in all experiments. As expected, the throughput decreases across all experiments in the DORAMI setup. Operations with smaller batch sizes show larger performance gaps due to more frequent context switches. Random writes with smaller batches perform significantly worse, compared to larger batch sizes. However for sequential writes, smaller batches initially perform better but eventually stabilize at around 1650 KB/s across all file sizes. This is due to the 512KB block size of the file system, which requires the kernel to read before writing data in smaller batches. This issue is particularly pronounced for random writes, where smaller batches are more likely to access uncached blocks.

Table 10: darkhttpd event counts for baseline (B) and DORAMI (D) execution. ocalls (row 3), total and normalized M-mode enters (rows 4-5), timer interrupts (rows 6-7).

Event		1KB		10KB		100KB		1MB		10MB	
		B	D	B	D	B	D	B	D	B	D
Ocalls	Total	11	11	11	11	11	11	11	11	20	20
M enters	Total	37.74	38.65	43.83	45.58	59.06	65.57	413	438.04	2686.83	3212.69
	Norm	37.74	38.73	4.28	4.45	0.57	0.64	0.40	0.43	0.26	0.31
Timers	Total	1.38	1.39	1.39	1.39	2.00	2.64	6.11	6.10	10.56	24.10
	Norm	1.38	1.39	0.13	0.13	0.02	0.026	0.006	0.006	0.001	0.002

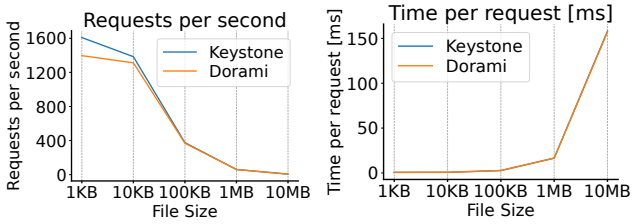


Figure 9: Darkhttpd: Requests per second and Time per request (latency) for Keystone and DORAMI, sizes 1 KB-10 MB.

Case-study 1: Webserver (darkhttpd). We demonstrate DORAMI's compatibility with real-world applications. Our first case-study is Darkhttpd, a webserver to serve html-based webpages [11]. We take darkhttpd version from Cerberus which is compatible with Keystone [52]. It executes the webserver in an enclave and serves webpages to the network using ocalls. We use apachebench to test webpages varying from sizes of 1KB to 10MB [3]. Fig. 9 shows the number of requests per seconds (RpS) and the time per request (TpR) for Keystone (baseline) and DORAMI for each file size. Only smaller file sizes show noticeable overheads, where RpS decreases by 13% and TpR by 15% for DORAMI. For a larger sizes (e.g., 1 MB), DORAMI overheads drop to 0.45%. It is almost 0 for 10 MB. Since RpS and TpR differences are only significant for smaller files, we conclude that DORAMI would not significantly impact SLAs. DORAMI does not impact ocalls (Tab. 10, row 3), the setup incurs 11 ocalls with 9 more for large file size to load more blocks. DORAMI incurs a fixed overhead for M-mode enters and timer interrupts. Tab. 10 shows that the total events increase with file size as expected—the larger the file,the longer it takes to serve it, resulting in more timer interrupts. When we normalize the number of enters and timers per unit of file size (1KB), we see fewer events for larger files, which is expected—larger files spend more time on I/O while either yielding CPU time slices or blocking the CPU execution for IO. Importantly, as the normalized events decrease, DORAMI overhead reduces.

Case-study 2: In-memory Database (SQLite). We measure a simple database server based on SQLite from Cerberus [52]. SQLite executes in an enclave and loads a database from the host using ocalls. As in Cerberus, we measure 1,000 `SELECT` queries and observe 5 % overhead compared to the baseline.

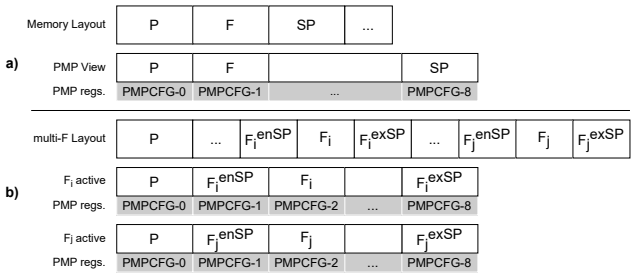


Figure 10: multiple-F: Memory layout and PMP configs. **P**=PMP compartment, **F**=Firmware compartment, **SP** =SallyPort, **enSP**=entry-SP; **exSP**=exit-SP. a) shows extract of DORAMI's memory layout and PMP config with a single F-compartment. b) shows the same for a multi-F implementation of DORAMI with two F compartments.

9 Discussion

We first discuss an approach for extending DORAMI from one to multiple F compartments. Then we outline security considerations for implementing ePMP in hardware.

9.1 Supporting Multiple F Compartments

DORAMI's design can be extended to create multiple F compartments, for example to isolate different firmware modules. Fig. 10(a) shows the current memory layout of DORAMI for supporting one P and one F compartment with the corresponding PMP configurations. In original DORAMI design we placed the code section of Firmware compartment directly after the code section of the PMP compartment for compartment transitioning (5.1). However this approach can not apply as-is to *n* compartments. We can solve this issue by placing a SallyPort Entry (enSP) and a SallyPort Exit (exSP) before and after each of the n Firmware compartments. Fig. 10(b) shows the memory layout for one P and multiple Fs. Such a solution would not need more PMP entries per compartment i.e., the number of needed entries does not increase with the number of compartments. Before transitioning into a particular F compartment (e.g., F_j), P can re-use an existing PMP configuration of another F (e.g., F_i) and configure it with F_j's configuration. This way, all other compartments automatically become inaccessible since they are not covered by any ePMP configuration. In total, compared to the single F design, such a solution would only require 1 additional PMP entry to be permanently reserved for all the F compartments.

9.2 Microarchitectural Considerations

Variations in ePMP implementation have security implications on DORAMI in two aspects: (a) location of the checks; (b) hazard management. All the ePMP open-source implementations that we analyzed (QEMU, Rocket, NOEL-V) adhere

to the above two requirements. We outline the detailed requirements, their security implications, and mitigations below for completeness.

First, the RISC-V specification does not define when or where PMP (or ePMP) access checks must occur within the CPU pipeline. Typically, these checks are performed during an instruction fetch phase for instruction addresses and during a memory access phase for data/operand addresses. Although the exact location of these checks is not critical for DORAMI, it is crucial that all PMP checks in M-mode are performed immediately and are never cached. For example, if manufacturers optimize checks by caching access permissions for M-mode in the TLB (despite M-mode's lack of memory translations), PMP changes require a flush to take effect. Specifically, DORAMI would require a TLB flush during compartment transition to apply the new PMP configurations. Second, when a PMP configuration is updated, DORAMI requires that the corresponding instruction should be marked as hazardous to ensure the pipeline is flushed after its execution. Without this step, instructions already in progress may execute under outdated PMP rules until the pipeline clears. The same issue exists in superscalar architectures; hazards should be synchronized across all pipelines to immediately reflect the new configuration. For consistency, PMP-related instructions should also act as instruction barriers, preventing reordering of subsequent instructions until the PMP update is fully applied. This ensures that no instructions beyond a PMP-related instruction are executed out of order. For DORAMI, PMP-related hazards are critical to ensure that the Firmware compartment F cannot execute any instructions outside of its boundaries (e.g. of the SallyPort). In summary, DORAMI assumes that apart from the core implementing ePMP correctly as per the specification, the implementation adheres to the above requirements as is the case in cores we evaluated.

10 Related Work

We discuss works beyond the ones covered in Section 2.4.

Use of PMP in TEEs. Keystone [53] is one of the first TEEs on RISC-V that solely leverages PMP features without any hardware modifications. SPEAR-V, Elasticlave, and Cerberus also leverage PMP features to establish secure enclave systems [52, 58, 64]. Timber-V, Penglai, Servas, Cure, and Sanctum introduce hardware modifications to achieve memory isolation through alternative means, such as bus-level or CPU-component-based isolation [36, 42, 50, 60, 62]. While these TEE implementations assume the entirety of firmware to be trusted and error-free, DORAMI serves as a complementary addition rather than a competitive alternative.

Confidential VMs. Modern TEEs offer a VM abstraction. In addition, they introduce a new privilege level that executes trusted software beneath the VM. For example, Arm CCA executes a security monitor in EL3 as part of the trusted firmware, but also introduces a Realm Management Monitor (RMM) that executes in the EL2 of realm world to isolate different realm VMs [5]. Similarly, Intel TDX has a TD module that executes in a new privilege level called SEAM Root Mode [15]. AMD SEV-SNP has Secure VM Service Module (SVSM) which optionally executes in VM Permission Level VMPL0 [2]. All of them use horizontal privilege separation.

Extension of DORAMI to Different Architectures. Commercial products like Intel SGX and TDX, or AMD SEV-SNP, offer confidential computing capabilities through enclaves and VMs [2, 15, 41]. These platforms, featuring fundamentally distinct memory isolation designs compared to RISC-V, implement proprietary measures for memory isolation, with closed-source firmware components. Similarly, Arm TrustZone [33] and CCA [5] introduce secure services and VM-based computing, respectively. TrustZone employs an Address Space Controller (ASC) [10] for memory isolation, while CCA introduces Physical Address Space (PAS) regions. Arm encounters challenges in privilege separation of code execution within the trusted firmware in EL3, equivalent to M-mode on RISC-V, which is 310 KLoC [6]. Future works can investigate if DORAMI approach can be applied to CCA.

RISC-V Hypervisor Extensions. SMMTT [27] isolates Physical Address Spaces for S-Mode software using memory tagging. SMMTT is a flexible alternative to Arm CCA [56], which also requires a universally trusted firmware in M-mode for configuration, similar to PMP. DORAMI complements SMMTT, PMP compartment can configure both SMMTT and ePMP, with remaining firmware in a separate compartment.

11 Conclusion

DORAMI is the first system that isolates the security monitor and the firmware on RISC-V. It uses an existing standard ISA feature, ePMP, to achieve this goal with minimal overheads. DORAMI is compatible with current RISC-V firmware & extensions and achieves reduction in the TCB. This can pave path for future works to formally verify the security monitor without reasoning about the rest of the firmware that executes alongside in the highest privileged execution layer on RISC-V.

Acknowledgments

We thank the Usenix Security 2024 reviewers for pointing us to NOEL-V and for their constructive feedback that significantly improved the paper. Thanks to Laurent Wirz for analyzing OpenSBI to identify the partition boundaries, Mélisande Zonta-Roudes for feedback on the early version of the paper and Supraja Sridhara for fruitful discussions on RISC-V TEEs. This work was supported by the Swiss Joint Research Center financed by Microsoft Research.

References

[1] ACE. https://github.com/IBM/ACE-RISCV.

[2] AMD SEV-SNP. https://www.amd.com/en/developer/sev.html.

[3] ApacheBench. https://httpd.apache.org/docs/current/programs/ab.html.

[4] Ariane Cpu. https://github.com/lowRISC/ariane.

[5] Arm CCA. https://developer.arm.com/documentation/den0125/0300/.

[6] Arm TF-A. https://www.trustedfirmware.org/.

[7] ARMv8 Registers. https://developer.arm.com/documentation/ddi0595/2021-12/.

[8] BerkeleyBL. https://github.com/andestech/BBL.

[9] Caliptra. https://caliptra.io.

[10] CoreLink TZC-380 Technical Reference. https://developer.arm.com/documentation/ddi0431/.

[11] Darkhttpd. https://unix4lyfe.org/darkhttpd/.

[12] Freedom U SDK. https://github.com/sifive/freedom-u-sdk.

[13] HEVD: kASLR + SMEP Bypass. https://fluidattacks.com/blog/hevd-smep-bypass/.

[14] Intel 64 and IA-32 Architecture Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[15] Intel TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[16] NOEL-V CPU. https://www.gaisler.com/noel-v.

[17] OpenSBI. https://github.com/riscv-software-src/opensbi.

[18] OpenTitan. https://opentitan.org.

[19] Oreboot. https://github.com/oreboot/oreboot.

[20] [PATCH 1/2] lib: sbi: fwft: check feature value to be exactly 1 or 0. https://lists.infradead.org/pipermail/opensbi/2024-June/007050.html.

[21] [PATCH] include: Adjust Sscofpmf mhpmevent mask for upper 8 bits. https://lists.infradead.org/pipermail/opensbi/2024-July/007147.html.

[22] [PATCH] lib: sbi: dbtr: fix potential NULL pointer dereferences. https://lists.infradead.org/pipermail/opensbi/2024-August/007184.html.

[23] [PATCH] lib: sbi: Fix timing of clearing tbuf. https://lists.infradead.org/pipermail/opensbi/2023-June/005078.html.

[24] [PATCH v6 12/12] lib: sbi: Fix missing '\0' when buffer szie equal 1. https://lists.infradead.org/pipermail/opensbi/2023-June/005170.html.

[25] RISC-V SBI Specification. https://github.com/riscv-non-isa/riscv-sbi-doc.

[26] RISC-V Specifications. https://riscv.org/technical/specifications/.

[27] RISC-V Supervisor Domains Access Protection. https://github.com/riscv/riscv-smmtt.

[28] RustSBI. https://github.com/rustsbi/rustsbi.

[29] RV8-Bench. https://michaeljclark.github.io/.

[30] SiFive U540 Platform. https://www.sifive.com/boards/hifive-unleashed.

[31] SiFive U74 Platform. https://www.sifive.com/boards/hifive-unmatched.

[32] Smepmp. https://github.com/riscv/riscv-tee/blob/main/Smepmp/Smepmp.pdf.

[33] TrustZone for AArch64. https://developer.arm.com/documentation/102418/0101.

[34] Xuantie CPU. https://www.xrvm.com.

[35] 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 *NDSS*, 2016.

[36] 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 *Usenix Security*, 2021.

[37] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Usenix NDSI*, 2008.

[38] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security*, 2004.

[39] Cerdeira, David and Santos, Nuno and Fonseca, Pedro and Pinto, Sandro. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *IEEE S&P*, 2020.

[40] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM. In *NDSS*, 2017.

[41] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016.

[42] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Usenix Security*, 2016.

[43] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory safety for low-level software/hardware interactions. In *Usenix Security*, 2009.

[44] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram S. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *ASPLOS*, 2015.

[45] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security*, 2013.

[46] Salvatore Di Girolamo, Andreas Kurth, Alexandru Calotoiu, Thomas Benz, Timo Schneider, Jakub Beránek, Luca Benini, and Torsten Hoefler. A RISC-V in-network accelerator for flexible high-performance low-power packet processing. In *ISCA*, 2021.

[47] Manuel Eggimann, Stefan Mach, Michele Magno, and Luca Benini. A RISC-V Based Open Hardware Platform for Always-On Wearable Smart Sensing. In *IWASI 2019*.

[48] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *OSDI*, 2006.

[49] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Usenix Security*, 2020.

[50] 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 *OSDI*, 2021.

[51] Norman Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Oper. Syst. Rev.*, 1988.

[52] Dayeol Lee, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanovic. Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing. In *CCS*, 2022.

[53] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *EuroSys*, 2020.

[54] Christian Lindenmeier, Mathias Payer, and Marcel Busch. EL3XIR: Fuzzing COTS Secure Monitors. In *Usenix Security*, 2024.

[55] Wojciech Ozga. Towards a Formally Verified Security Monitor for VM-based Confidential Computing. In *HASP*, 2023.

[56] Ravi Sahita and Atish Patra and Vedvyas Shanbhogue and Samuel Ortiz and Andrew Bresticker and Dylan Reid and Atul Khare and Rajnesh Kanwal. CoVE: Towards Confidential Computing on RISC-V Platforms. In *arXiv*, 2023.

[57] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *Usenix Security*, 2022.

[58] David Schrammel, Moritz Waser, Lukas Lamster, Martin Unterguggenberger, and Stefan Mangard. SPEAR-V: Secure and Practical Enclave Architecture for RISC-V. In *ASIA CCS*, 2023.

[59] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, and Binyu Zang. Deconstructing Xen. In *NDSS*, 2017.

[60] Stefan Steinegger, David Schrammel, Samuel Weiser, Pascal Nasahl, and Stefan Mangard. SERVAS! Secure Enclaves via RISC-V Authenticryption Shield. In *ESORICS*, 2021.

[61] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE S&P*, 2010.

[62] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *NDSS*, 2019.

[63] Kenichi Yasukata, Hajime Tazaki, and Pierre-Louis Aublin. Exit-Less, Isolated, and Shared Access for Virtual Machines. In *ASPLOS*, 2023.

[64] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E. Carlson, and Prateek Saxena. Elasticlave: An Efficient Memory Model for Enclaves. In *Usenix Security*, 2022.