Miralis and the hardware burger

Miralis and the hardware burger

Projet de Master - Masterarbeit

François Costa

Kicking out the firmware from the trusted computing base and lightweight formal verification of the virtual firmware monitor.

Supervisors:
Charly Chastes, Edouard Bugnion, Timothy Roscoe

Date: March 28, 2025

Abstract

The role of firmware has evolved significantly over the past decades. Beyond discovering, initializing, and monitoring the system's chipset, board, and devices, firmware now serves as the root of trust and plays a crucial role in confidential computing. However, vulnerabilities in non-security-critical components of firmware have repeatedly led to compromises in the system's core TCB.

We propose an alternative architecture that excludes the non-security-critical part of the firmware from the TCB by isolating it within a virtual machine with the introduction of a simple *virtual firmware monitor*. We leverage the formal ISA specification to verify some of its key critical components using lightweight formal methods.

We present Miralis, the first virtual firmware monitor. Miralis successfully boots Linux on two RISC-V platforms: the VisionFive2 board and the Premier P550 board, with a virtualized OpenSBI on RISC-V. Furthermore, Miralis can isolate the operating system, confidential virtual machines, and enclaves using flexible policies. Through its construction, we demonstrate that RISC-V's M-mode satisfies the Popek & Goldberg criteria for classical virtualization. Our evaluation shows that Miralis eliminates vendor-provided, platform-specific firmware from the TCB with no measurable performance impact on real-world applications.

Acknowledgements

I would like to first thanks Charly Castes, Edouard Bugnion and the DCSL – Data Center Systems Laboratory for giving me the opportunity to pursue a master's thesis here. I had six excellent months. The conclusion of my academic journey could not have ended under better circumstances.

Next, I want to thank Timothy (Mothy) Roscoe for his supervision, he gave useful suggestions during the entire thesis. It was a great pleasure to benefit from his strong knowledge and experience in Operating systems.

One person who was more discrete but has nonetheless contributed significantly to making my daily experience enjoyable is Margaret Church. She oversaw all administrative aspects throughout my thesis and consistently ensured that I had a weekly time slot with the professor, despite his busy schedule. We had many interesting discussions which I am going to miss.

Additionally, I would like to extend my thanks to my former mentor Lazar Cvetkovic and professors Michal Friedmann and Ana Klimovic, who gave me the possibility to start systems research. I had the opportunity to work on *Dirigent*, a cluster manager that was presented at *The 30th Symposium on Operating Systems Principles (SOSP 24)*.

Lausanne, 30 March 2025

Contents

List of Figures

List of Tables

1 Introduction

1.1 The firmware: historical perspective

The role of the firmware within computer systems has evolved significantly over the past few decades. Historically, the firmware was primarily responsible for the initialization of hardware components, such as chipsets, motherboards, peripheral devices and provides management services at runtime. For example, the INT 13h interrupt in BIOS mode provide low-level disk services such as reading and writing on disks, retrieving the available memory map or writing characters to the disk. Additionally, interrupts INT 15h offers a broader set of system services, going beyond disk operations to include extended memory services and system hardware management and Interrupts 10H is responsible for video an display management. Figure 1.2 shows an example where a x86 bootloader calls the firmware to communicate with the hardware [39]. These routines are available in *real mode* on x86 processors. *Real mode* is the initial operating mode of x86 CPUs, originating from the Intel 8086, where the processor runs in a simple 16-bit environment. In this context, we observe that the firmware had a single role: *managing hardware*.

1.2 A tension between two worlds: hardware management and security code

A few decades later, the firmware adapted to the increasingly complex demands of modern technology and became a critical component in enforcing security invariants due to its particular position in the trusted computing base. This new role emerged with the growing popularity of confidential computing [90, 33, 99, 48, 101, 63], where the firmware plays a security-critical role in enforcing strict isolation policies to protect trusted execution environments (TEEs). The firmwares such as BIOS, UEFI, and Opensbi operate at the highest privilege levels within a CPU (M-mode in RISC-V, EL3 in ARM, and SMM in x86), making them a security-critical software layer. Unfortunately, the primary hardware management part of the firmare conflicts with the goal of confidential computing. Hardware management code requires large and

```
print_string:
    pusha
    next:
    mov al, [bx]
    cmp al, 0
    je  continue_in_next
    mov ah, 0x0e
    int 0x10
    add bx, 1
    jmp next
    continue_in_next:
    popa
    ret
```

Figure 1.1: Low level routine from a x86 bootloader [39] relying on the firmware to interact with the hardware

complex codebases, naturally increasing the risk of security vulnerabilities and the position of the firmware in the TCB makes the firmware an interesting target for exploitation.

Firmware attacks can have severe consequences, including bypassing secure boot, leaking secret keys, confidential information, and achieving a full access of the system. Despite the firmware's critical role within the *trusted computing base* and its status as a high-value target for attackers, minimal effort has been made to reduce its attack surface. The dual responsibilities of the firmware—managing complex hardware interactions and enforcing security invariants—present a fundamental tension[30]. On one hand, effective hardware management often necessitates a substantial codebase, which can introduce a variety of vulnerabilities. On the other hand, maintaining strict security requires transparency, a minimal code footprint, and robust correctness guarantees. A non-trivial part of the TCB contains non-security-critical code that does not need to be tightly integrated with the security features of the system. In addition, despite its central importance, this low-level software is often distributed as opaque binary blobs, which obscure the underlying code and complicate efforts to evaluate its security. In the extreme case, a technical report describing the vendor firmware of the Cavium ThunderX Arm CPU reports above a million lines of C, including a custom standard library, a scheduler, and a Lua interpreter to customize firmware operations. Such extensions drastically increase the TCB and weakens the security of all software running on the machine, despite the dubious need for some of these functionalities.

1.3 Miralis: a security monitor that isolates the firmware

Castes et al. [15] proposes the concept of a *virtual firmware monitor* and the Miralis hypervisor. They remove the firmware from the root of trust by introducing a clear separation between

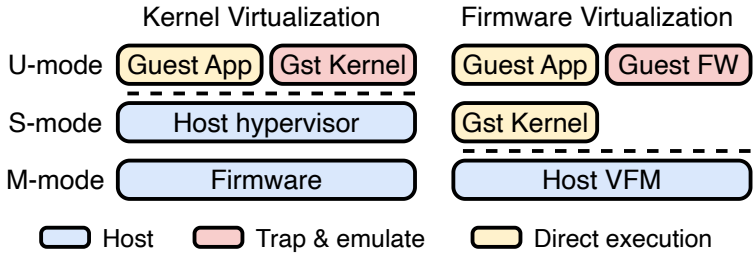

Figure 1.2: Comparison of classical virtualization for kernels (S-mode) and firmware (M-mode).

security-critical code and hardware management primitives. This approach draws inspiration from microkernel designs [1, 12, 43, 64, 91, 113, 65, 84] and the evolution from Type-1 to Type-2 hypervisors. Microkernels execute essential operating system functions such as memory allocation, paging, and process scheduling in deprivileged user mode while the core handles security-sensitive operations. Miralis adopts a similar principle by running the firmware directly in user space, shifting hardware management code to user space. It relies on Popek and Goldberg virtualization to emulate privileged instructions. When a privileged instruction is executed, Miralis traps and emulates the instruction via the virtual the firmware monitor. Miralis runs unmodified, sandboxed the firmware alongside real-world Linux images with minimal overhead.

1.4 Contributions of this work

Castes et al. [15] presents a prototype of a *virtual firmware monitor* named Miralis. Miralis alone *virtualises* the firmware but doesn't *isolate* it from the rest of the system. For example, the firmware has access to the entire memory space. While it can't directly jump to machine mode (M-mode) and gain the highest privileges, it can still modify the text or data segments of the operating system and confidential enclaves, enabling a range of attacks. With access to program memory, an attacker can manipulate existing code snippets, inject malicious code, or read/write unauthorized memory. Additionally, the firmware can corrupt data by altering variables or function arguments, modifying program behavior, or leaking sensitive information such as cryptographic keys and passwords.

This thesis starts by filling the missing requirements for a complete and efficient virtualisation of the firmware in Miralis. It does so by adding support for the *hypervisor extension*, interrupts virtualisation. On top of that if fixes a series of bugs and add the necessary features that allowed Miralis to run on the Premier P550 board.

Then the thesis introduces *security policies*, a component responsible for *enforcing* isolation in Miralis, as merely isolating the firmware in its own context is insufficient. Miralis supports various security policies, each isolating the firmware from specific components. This work explores the Strict Protect Payload Policy, which enforces strict isolation between the firmware

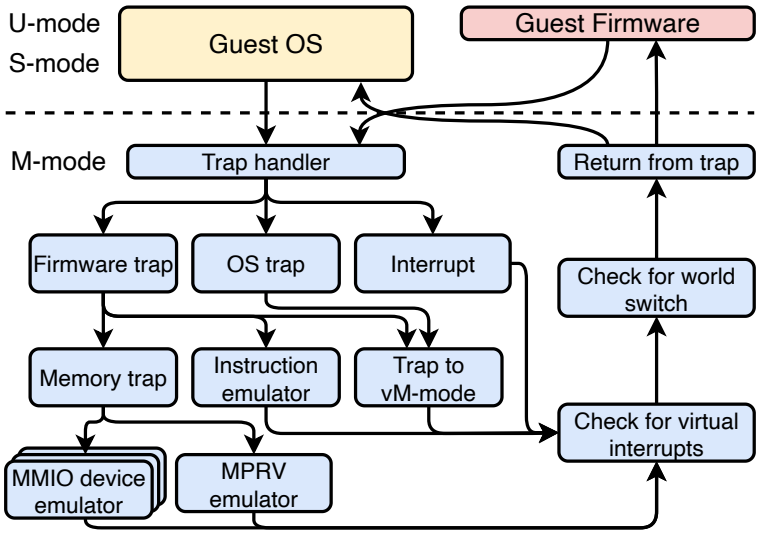

Figure 1.3: A high level overview of Miralis

and the payload, and the Ace Policy, which protects confidential virtual machines from the firmware.

Next, we leverage lightweight formal methods [51, 2, 10] to verify that 40% of the codebase exhibits a behavior consistent with the official RISC-V specification. This extends Virt-sail, a proof-of-concept Sail-to-Rust compiler, into a fully functional compiler that transpiles a subset of Sail code to Rust. We then leverage the generated Rust code to formally verify the trap and emulation mechanism, as well as the virtualization of interrupts, traps, and PMP register configuration.

Finally, this thesis rigorously evaluates Miralis through extensive performance tests on the VisionFive2 board and the Premier P550 board, including CPU, disk, and network microbenchmarks, as a performance through a range of real world workloads such as compiling the Linux kernel and performance evaluation of three databases, namely Memcached, Redis and MySQL. We conclude the benchmark section with a measurement of Keystone in Miralis. Across all workload, this work shows that Miralis successfully runs an unmodified the firmware with no runtime cost compared to the Opensbi open-source the firmware.

- Support for the Premier P550 board board
- Introduction of security policies, including two specific policies for isolating the payload and confidential virtual machines
- Development of a Sail-to-Rust compiler and lightweight formal verification of the firmware hypervisor
- Rigorous evaluation of Miralis
- An accepted paper at HotOS XX
- Contribution to a submission at SOSP25

2 Background

2.1 Virtualization

In their seminal work, Popek and Goldberg established criteria to determine whether an instruction set architecture (ISA) can be virtualized effectively. The sufficient criterion for an ISA to be considered virtualizable is that every *sensitive instruction* must also be a *privileged instruction*. *Sensitive instructions* modify the system's configuration or depend on it to function, while privileged instructions trigger a trap when executed in a less privileged mode. Virtualization relies on the *trap-and-emulate* mechanism: privileged instructions executed at a lower privilege level cause a CPU exception, allowing the hypervisor to emulate their behavior.

We consider RISC-V ISA to illustrate the *trap-and-emulate* emulation. The *csrrw* instruction, which writes a value to a CSR and stores the old CSR value into a general-purpose register, is a sensitive instruction as it interacts directly with the configuration of the system. Conversely, a divide instruction is not sensitive because its execution is independent of system configuration. When a virtualized OS executes a divide instruction, it runs with no performance penalty. However, executing a *csrrw* instruction raises an exception, requiring hypervisor intervention. Although this introduces some overhead, the proportion of sensitive instructions is statistically insignificant, making *trap-and-emulate* virtualization efficient.

Architectures meeting Popek and Goldberg's criteria are termed *classically virtualizable*. These architectures support *trap-and-emulate* techniques, where sensitive instructions trigger traps handled by the hypervisor. *Non-classically virtualizable* architectures require more complex and resource-intensive software techniques. The x86-32 architecture is an example of a nonvirtualizable ISA. It fails virtualization criteria because certain sensitive instructions are not privileged. For instance, the *popf* instruction, which manipulates the flags register and reveals the state of the interrupt enable flag, is sensitive but not privileged. This violates Popek and Goldberg's criteria, making x86-32 nonvirtualizable in the classical sense. VMware's Virtual Machine Monitor [13] addresses x86-32's nonvirtualizability through dynamic binary translation, dynamically rewriting sensitive non-privileged instructions into virtualizable

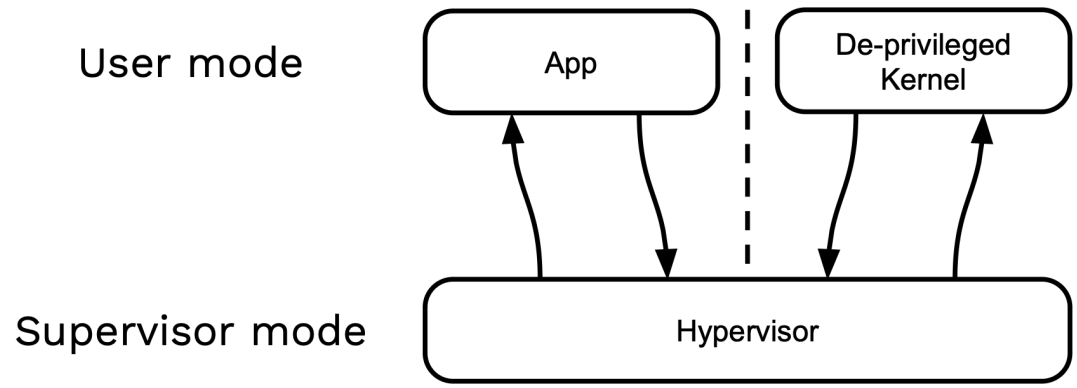

Figure 2.1: *Trap-and-emulate* virtualization

sequences at runtime.

2.2 Vendor firmware

In addition to selling hardware, CPU vendors develop and maintain the firmware required to operate the SoC. This firmware consists of two main components: a transient boot-stage software that handles CPU initialization, DDR training, the loading of subsequent stages, and a resident software that runs at runtime to manage the SoC and support OS functionality. Unless stated otherwise, in this thesis, we refer only to the resident software running on the application CPU as *firmware*. Software executed on auxiliary cores, accelerators, and peripheral devices falls outside the scope of this work and is addressed in separate research efforts [34, 35].

2.3 Cisc and Risc platforms

This thesis focus on RISC platforms, specifically RISC-V, while acknowledging that the findings are applicable to other RISC architectures as well. We exclude CISC platforms, for two key reasons: (1) their extensive dependence on microcode for system management [6], and (2) the complexity of their privilege modes, which are vendor-specific, often undocumented, and required for system operations—such as SMM, SEAM, Architectural Enclaves, or the AMD Secure Processor [18, 25, 90]. In contrast, RISC platforms handle system management through software and provide a clearly documented hierarchy of privilege levels.

2.4 Firmware vulnerabilities

Firmware is a potential attack vector and represents an interesting security threat due to its privileged position in the software stack. It serves as the root of trust, linking software and

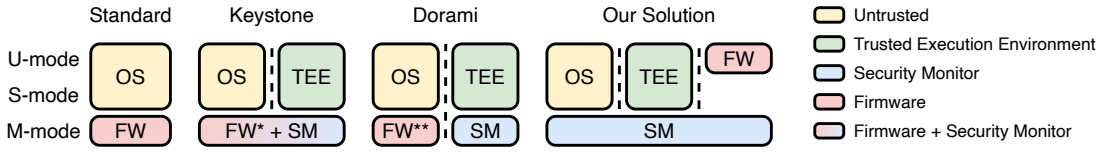

Figure 2.2: Comparison of TEE deployments on RISC-V. With Keystone [101] and most existing TEEs the security monitor is co-located with the vendor firmware. Dorami [57] is a recent attempt at privilege separation, but requires firmware refactoring and binary scanning. Miralis achieves privilege separation with no firmware modification. FW*: lightweight firmware modifications, FW**: intrusive firmware modifications.

hardware. If an attacker gains control of the firmware it gains unrestricted access to the entire system. Previous attacks have exploited vulnerabilities in firmware code, with researchers identifying various attack vectors, including side-channel attacks, modification attacks, and firmware-specific vulnerabilities.

The PKFAIL vulnerability [77] exploits cryptographic flaws in PKI implementations of TLS and SSL protocols, allowing attackers to bypass security mechanisms by manipulating public key generation and validation. The Sinkclose vulnerability [104] leverages AMD's legacy compatibility features to manipulate the TClose function, redirecting privileged SMRAM memory and granting attackers unauthorized system control at the SMM level. The Boot Unguarded attack [9], following the 2023 MSI data leak exposing Intel's BootGuard private key, bypasses secure boot processes, enabling rootkits and highlighting supply chain risks.

Additionally, AMD Ryzen processors [3] suffered from an fTPM stuttering bug, where SPIROM latency caused system-wide stuttering, mitigated through BIOS updates. The LogoFAIL vulnerability [54] exploits outdated UEFI firmware image parsers, using buffer overflows during boot to bypass Secure Boot protections. BYOBD attacks [7] exploit vulnerable drivers to disable critical defenses like Secure Boot, as demonstrated by the BlackLotus UEFI bootkit. Finally, the Three Flaw Secure Boot Bypass [76, 70, 71] involves signed third-party UEFI bootloaders, allowing attackers to bypass Secure Boot, install backdoors, and tamper with OS files, emphasizing the persistent challenge of securing boot integrity. Recently, Google Security Team discovered a vulnerability in AMD Zen 1-4 CPUs [100] that allows an attacker with local administrator privileges to load malicious microcode patches due to an insecure hash function used in signature validation.

2.5 RISC-V security monitors

Several security monitors have been developed for the RISC-V architecture, each contributing to system security through using hardware and software protection mechanisms. Figure 2.2 shows a subset of them. Keystone [101] is an open-source trusted execution environment (TEE) framework enabling secure enclaves on RISC-V processors, isolating sensitive data from malicious software and system administrators while maintaining a minimal trusted

computing base. Sanctum [58] prevents software-based side-channel attacks by isolating enclaves within the processor and ensuring that even privileged software, such as hypervisors, cannot compromise sensitive data.

ACE [75] enhances confidential computing by enforcing strict security policies between *virtual machines* and *confidential virtual machines*. It employs hardware-based safeguards to protect critical operations, such as boot code integrity and memory isolation. Dorami [57] focuses on privilege separation within TEEs, securing transitions between privilege levels and protecting critical memory regions.

PHMon [27] provides a flexible, programmable hardware monitoring system for RISC-V processors, enforcing security policies through real-time hardware integration to detect violations. R5Detect [11] enhances control-flow integrity by detecting and preventing control-flow attacks on RISC-V architectures. Tyche [14] introduces a security monitor exposing a low-level API for building and composing isolation abstractions, rather than relying on vendor-specific extensions, improving security and interoperability.

2.6 Lightweight formal methods

The main difference between formal methods and lightweight formal methods [51, 2, 10] lies in their approach. Unlike traditional formal methods, lightweight formal methods aim to verify specific properties of a system pragmatically, without requiring the exhaustive and resource-intensive process of full formal verification. This approach focuses on automating checks for key properties that are critical to the system's functionality, making it feasible for systems under active development, where both the software and its specifications are constantly evolving.

While full formal methods seek to rigorously prove the correctness of an entire system through comprehensive formalization and proof, lightweight methods focus on partial formalization. As a result, they offer an excellent trade-off between the security guarantees found and the issues detected in the system, relative to the time spent on verification.

3 Wrapping up the Miralis hypervisor

Castes et al. [15] produced a nearly complete version of Miralis that adheres to Popek and Goldberg's criteria for virtual machines [78]. Therefore, zhe first milestone of this thesis focuses on finalizing the Miralis hypervisor

3.1 Emulation of hypervisor extension

The Hypervisor extension [37] (H-mode extension) is an optional extension introduced in the official RISC-V specification. This extension improves resource utilization in virtualized environments. Virtualization is nowaday everywhere and therefore Miralis must support this extension as well.

Emulating the hypervisor extension is relatively straightforward, as it requires only a few extra CSR registers and two memory fence instructions. Table 3.1 lists the new registers. Most of the logic introduced by the extension occurs in lower-level privilege modes, where, when a guest operating system attempts to write to an s-mode register, the underlying h-register is written instead. Hardware logic running in modes other than M are orthogonal to Miralis. In Miralis, we extend the virtual context by adding these new registers and saving/restoring their values during context switches. Since the extension is optional, Miralis stores and restores the CSR only when the extension is explicitly present in the MISA register. On top of that, there are a few minor modifications to the usual CSRs when the hypervisor extension is present, such as Mstatus or MtVal2. These changes are minimal, and Miralis emulates them as well.

Next to the introduction of a few registers, we emulate hypervisor memory-management fence operations to maintain the correct ordering of memory operations in the virtualized environment. Hypervisor memory-management fence instructions run either in M-mode or HS-mode operating system and, therefore, will trap when executed in VM mode. We add support for these instructions in the Miralis decoder and simply issue the equivalent instruction from Miralis itself when the M-mode issues on of the two instructions.

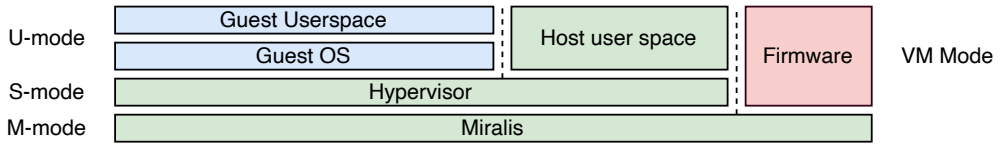

Figure 3.1: Hypervisor extension layout

Category	Registers
Control and Status Registers	hstatus, hcounteren
Trap Setup Registers	hedelg, hideleg
Trap Handling Registers	htval, htinst, hgatp
Virtual Interrupt Registers	hip, hie
Timer Registers	hgeip, hgeie
Guest Registers	vsstatus, vsie,vstvec, vsscratch, vsepc, vscause, vstval, vsatp

Table 3.1: Registers introduced by the hypervisor extension

3.2 Timer interrupt virtualisation

The firmware now runs in userspace, meaning it no longer has a direct way to manage the machine timer interrupt, as this mechanism is reserved for M-mode. Miralis therefore virtualizes two components to grant the firmware access to the machine timer interrupt. Miralis tracks and intercepts calls when the firmware requests the CLINT to alter the mctimecmp register. This logic is implemented by protecting the address of the `mtimecmp` register with a PMP entry, catching the access fault, and then injecting the request into the physical CLINT driver as a second step.

The second aspect that needs to be virtualized is the forwarding process from Miralis to the firmware. When Miralis receives a timer interrupt, it forwards it to the firmware following algorithm 1. It adjusts the MPP field, updates the mcause register, writes the current program counter, sets MtVal to zero, and then jumps to the firmware. Currently, Miralis doesn't use interrupts, so there is no need to multiplex the machine timer interrupt between the firmware and Miralis itself. In section 5.4, we extend this logic to multiplex the machine timer interrupt with the machine software interrupt.

3.3 Detection of optional extensions at runtime

The *misa* (Machine ISA) register is a RISC-V M-mode register, used by the hardware to encode information about the availability of particular extensions. Currently, Miralis determines at compile time which extensions are present. This approach forces Miralis to be deployed on a platform that exactly matches the extensions enabled at compile time. In practice, running Miralis under the assumption of a perfect match between Miralis and the hardware

Algorithm 1 Interrupt virtualisation procedure

1: Mstatus.MPP ← `current mode`
2: MPIE ← $Mstatus.MIE$
3: Mstatus.MIE ← 0
4: Mcause ← next_int
5: Mepc ← PC
6: Mtval ← 0
7: Mode ← Machine ▷ When Moving to the firmware Miralis changes the mode to User
8: PC ← *set_pc_to_mtvec*()

is inconvenient and leads to silent bugs. This motivates the introduction of a component that adapts Miralis to the underlying hardware. We do this by determining at *runtime* which extensions are available. During the initialization phase, Miralis reads the contents of the MISA register and updates the Miralis virtual context accordingly. Furthermore, Miralis simplifies its implementation by making the MISA logic read-only and zero, thus disabling the deactivation of optional extensions. This behavior aligns with the official Sail RISC-V specification, as MISA can be configured to be *read-only*, making it a legal optimization.

3.4 Implementation of abstractions for the PMP registers

The Physical Memory Protection (PMP) registers are a RISC-V memory protection mechanism designed to prevent access from supervisor and user mode to specific areas of memory. This is similar to segmentation in the x86 ISA.

In the context of Miralis, PMP registers protect Miralis itself, virtual devices, and a few entries allocated for *security policies* 5. Miralis organizes the PMP entries into a static layout, which is determined at compile time based on the number of virtual devices and the associated protection policies in the system. Both the Strict Protect Payload Policy and Ace Policy policies allocate two slots, while a virtual device typically requires one *NAPOT* PMP entry. Since the PMP entries are matched in priority order, we place them before the virtual PMP entries of the firmware. This ensures that Miralis maintains full control over the entire memory in any configuration, while delegating part of the memory to the firmware. Figure 3.2 shows the layout of the PMP entries in Miralis.

A non-negligible consideration when modifying the PMP entries is the cost of flushing the TLB, which can become a performance bottleneck. Since flushing the TLB is typically an expensive operation, Miralis only flushes the TLB when absolutely necessary. Each time Miralis changes a PMP register, it sets a flag indicating that the state has been modified. Miralis will flush the TLB when the state is modified, before transitioning to either to the firmware or the payload.

Finally, setting up the PMP registers can sometimes lead to confusion because *NAPOT* (Natural Power of Two) and *TOR* (Top of Range) entries have specific formats and constraints that are not intuitive. *NAPOT* entries refer to memory regions with natural power-of-two sizes.

Entry	Region	Description
1	All Catch	Catching all addressses
2	Miralis Core	Protection for Miralis core memory region
3	Virtual Devices	Protection for virtual devices memory regions
4	Protection Policy	Enforcement of the memory protection policy
5	Empty Entry	Reserved space, no assigned region
6	VM-mode firmware	Protection for virtual machine firmware memory
7	Last Entry	Final PMP entry, used for alignment or reserved

Table 3.2: PMP layout in Miralis

These can be tricky to work with because typical systems expect sizes that are powers of two (e.g., 2, 4, 8 bytes). Specifying non-aligned memory regions can be confusing, as users must manually account for non-standard sizes and boundaries. *TOR* entries, on the other hand, are for memory regions with sizes that are powers of two, but these sizes must be divisible by 4. This can be an unexpected restriction for the programmer. For example, a 16 KB region is valid, but a 20 KB region is not, even though 20 is a power of two.

To make the PMP code less error-prone and easier to read, we introduce two helper functions that simplify working with *NAPOT* and *TOR*. These functions take a size and an offset argument to construct the corresponding PMP entries in Miralis.

Function	Effect
Convert NAPOT to Size	Adjusts non-standard sizes for correct processing.
Convert TOR to Size	Ensures power-of-two sizes are also divisible by 4.

Table 3.3: Helpers functions for the PMPs

3.5 Spike

Spike [103] is the official open-source, software-based RISC-V emulator. It is used to emulate the hardware behavior of a RISC-V processor, as specified by the official Sail implementation [83]. It helps developers run and test RISC-V programs without needing to run the actual binary blob on real hardware. While Qemu is fast when emulating RISC-V due to its use of *dynamic binary translation* and a *tiny code generator* acting as a lightweight JIT compiler, Spike focuses on precision. It simulates instructions with a higher degree of accuracy. For example, Spike updates the performance counter, whereas Qemu does not.

Spike offers other benefits, such as the ability to disable misaligned loads and stores, which allows us to test the emulation of this logic without requiring real hardware. We extend the configuration and test infrastructure so that it is possible to run workloads on both Qemu and

Spike. A user can now specify "Spike" as the platform instead of "Qemu" in the configuration file, and the runner will execute Miralis on the corresponding platform.

To summarize, we use Spike in Miralis for:

1. **Testing on a second emulator with different characteristics:** Spike allows us to test our firmware on a second emulator with different characteristics. Spike can disable the emulation of misaligned loads and stores, enabling us to test our code in more realistic scenarios.
2. **Precise performance counter updates:** Spike updates performance counters with higher precision, making it an ideal platform for measuring the overhead of Miralis. This allows us to assess the impact of changes without needing real hardware for every commit.

3.6 Test device

A *virtual device* is a software-based representation of a physical hardware device, such as a keyboard, mouse, timer interrupt, or more specialized hardware like a GPU. Instead of using actual physical components, a virtual device *emulates* the behavior and functionalities of a hardware component through an equivalent software interface. It allows users to interact with and test applications or firmware in a controlled, simulated environment without the need for the actual hardware, and enables additional functionalities such as *multiplexing the actual device*, removing the need to have multiple devices.

Memory-mapped devices consist of volatile registers. Compared to traditional memory, the value of these registers is *non-deterministic* and changes over time. Testing such a virtual device interface is more challenging due to this non-deterministic behavior. To address this, we created a specific virtual, *side-effect-free* test device used solely for testing purposes. It emulates a remote key-value store interface and is used to assert the correctness of the virtual device interface in Miralis. During the creation of this test device, we attempted to read and write the well-known value *0xdeadbeef* to the device, which led to the discovery of a bug in the virtual context, resolved by the commit: *Bugfix: sign extension on device load is sometimes broken.*

3.7 Improving the UART driver on the VisionFive2 board

The UART of the VisionFive2 board required some refactoring and was not optimal in terms of performance. After writing to the *serial_port_base_addr* register in UART, Miralis was running one million *nops*. The classical spinning implementation was not working, and the driver was stuck. After analyzing the root of the problem, it was found to be related to the width of a register, which was 4 bytes instead of 1 byte. The solution was to extend the driver slightly and add support for 4 bytes registers.

3.8 Software interrupt virtualisation

In a RISC-V system, *Inter-Processor Interrupts* (IPIs) is a mechanism used to send interrupts between different cpu cores within a multi-core system. IPIs are used for synchronization, signaling, or communication between processors, allowing them to coordinate tasks or handle events that require attention across the cores. To trigger an IPI in RISC-V, a core writes to mip register and sets the interrupt pending bit for another core. The interrupt is then delivered to the target core based on its interrupt handling logic and configuration, typically via the interrupt controller. This mechanism allows for efficient inter-core communication and synchronization in parallel processing environments.

Since the Miralis and the firmware might send IPI to other cores, virtualising the `machine software interrupt` is a bit more elaborate. We separate the clint into two components, `the virtual clint` and `the physical clint`. The physical clint is reponsible for propagating the signal we want to send *physically*. On top of that we have a virtual clint where we have have two additional array of Atomic boolean values. The first array corresponds to the machine software interrupt triggered by the firmware and the second array correspond to the machine software interrupt request by the policy module in section Miralis 5. If Miralis or the firmware triggers a `machine software interrupt`, the virtual clint first mark the boolean array and then triggers the physical clint. The second core can then demultiplexes the received values and decides to which components it should forward the `machine software interrupt`. That way we successfully virtualize the machine software interrupt.

3.9 Optimising Miralis

The *hot path* corresponds to the sequence of operations taken during a world switch or a virtual firmware trap. This section of the code represents the overhead introduced by Miralis, as it is executed after each exception. This code should run as quickly as possible to minimize the overhead imposed by Miralis. Using the *tracing_firmware*, we can precisely measure the number of cycles consumed by Miralis, which helps us optimize the *hot path*. We identified five key areas where optimizations had an impact on latency.

First, we improved the benchmark infrastructure that was being used to take measurements during trap handling, which involved approximately seven locks in each critical path. Eliminating this infrastructure reduced the latency by about 1000 cycles in the context of a firmware trap. Second, we moved the log filtering check to compile-time. This change reduced the runtime overhead by about 100 cycles during a firmware trap, with additional savings during a world switch. Third, we enabled release mode, which activates many compiler optimizations and disables debugging-specific features, such as runtime checks and extensive logging. This significantly improved both performance and resource utilization. Fourth, we made additional assumptions in the decoder: when Miralis receives a trap, it can automatically extract implicit information about the trapping instruction. For example, it knows that the last 7 bits

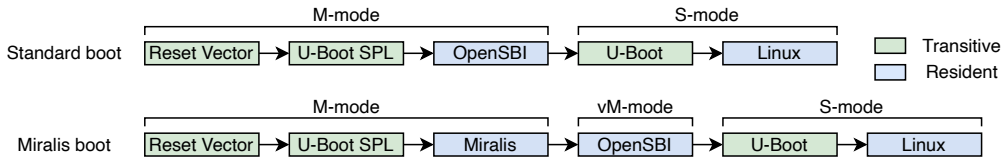

Figure 3.2: Boot Flow

correspond to the system opcode in the case of an illegal instruction, or it can directly extract the registers in the case of a misaligned load, without needing to decode the load instruction again. This optimization resulted in a gain of 20 cycles during a firmware trap and 70 cycles during a world switch. Finally, we removed an unnecessary lock protecting the CLINT, which had been causing contention during I/O-intensive workloads. The CLINT driver is accessed frequently, and the Miralis lock was redundant because the hardware serializes automatically read and writes to the same memory location. The equivalent Opensbi implementation is also lock-free, further justifying this change.

3.10 Engineering effort in the CI/CD pipeline

Miralis is security-sensitive due to its privileged location in the software stack. Therefore, we maintain high development standards, one of which is the presence of a robust CI/CD pipeline. We ensure that Miralis passes all tests at every commit. In addition to a code formatter, we introduced two new tools into the pipeline: a linter and a cargo deny tool. The linter checks the code for common issues, while cargo deny scans the content of external crates to flag any security concerns. We use Clippy as the primary linter for Rust, as it is the most widely used and effective tool for this purpose.

3.11 Introducing U-Boot

Without operating system, Miralis consists solely of its firmware and a virtual firmware emulator. Previous work [15] involved executing a base Linux kernel as an embedded payload within Opensbi. In this work, we extend the boot flow by introducing U-Boot in S-mode operating system, aligning more closely with the standard boot process. This standard procedure typically involves three key components: U-Boot SPL, Opensbi, and U-Boot, which together load the Linux kernel, as shown in the first half of Figure 3.2.

In this setup, Opensbi loads U-Boot, which then boots the Linux kernel. U-Boot is the open-source bootloader that comes with the Linux kernel. It plays a crucial role in the boot process by searching for and loading the Linux kernel. It can handle different types of booting processes, such as booting from flash memory, network booting, or booting from external devices like SD cards, and is highly customizable.

In this work, we embed U-Boot within the unmodified firmware and introduce an additional

state transition between Miralis and the Linux kernel. Instead of the traditional Miralis → Opensbi → Linux kernel flow, the sequence now becomes Miralis → Opensbi → U-Boot → Linux kernel. In this process, Miralis runs prior to Opensbi, which then transitions to U-Boot in S-mode operating system. The firmware operates in VM-mode throughout the flow. This approach allows us to place the Linux kernel on a disk, and U-Boot will handle the loading at runtime.

By default, on platforms like Qemu and Spike, the memory map installs Opensbi at 0x8000000 and U-Boot at 0x80200000. In Miralis, however, the firmware is virtualized and runs at 0x80200000, while U-Boot is relocated to 0x80400000. To accommodate this, we patch U-Boot by adjusting its text start address, leaving Opensbi unmodified, as it can execute from any memory location. On physical hardware, we maintain the original memory layout by loading Miralis after Opensbi and U-Boot. This approach avoids additional patching and recompilation, ensuring compatibility with the standard boot process.

Adding U-Boot to the CI/CD pipeline is a natural choice due to its importance in the boot process of a Linux distribution. Similar to the "normal" boot flow, we use U-Boot as the next step in the boot process after Opensbi. We compile a second modified version of U-Boot that triggers a shutdown upon entering the user shell. The compiled binary blob is used only for the CI/CD pipelines.

3.12 Running Linux distributions in Qemu / Spike

The Linux kernel is the core of any Linux-based operating system. It manages hardware resources (CPU, memory, and devices) and provides essential services that allow higher-level software to interact with the hardware. The kernel is responsible for tasks such as process scheduling, memory management, device drivers, and file systems. Linux distributions, such as Ubuntu, Fedora, OpenSUSE, and Debian, are built around the kernel and include not only the kernel itself but also a wide array of software packages, including desktop environments, utilities, system libraries, and applications, offering a complete environment for running applications and performing tasks like browsing, word processing, and more.

We have added the ability to pass a disk image directly to the Linux kernel in the runner configuration, allowing Miralis to run not only the basic Linux kernel but also multiple Linux distributions. We have successfully run the unmodified RISC-V versions of Ubuntu [66], Fedora [79], openSUSE [19], and Debian [21] in this setup.

For example, the runner downloads the official Ubuntu image from the official website and extracts it. Following the guidelines provided in the official Ubuntu RISC-V documentation, Qemu is configured for the Ubuntu distribution by setting up a virtual network interface using the *virtio protocol* [73]. This interface connects to a user-mode network backend, enabling internet access through *NAT*. Additionally, a *virtual random number generator* (RNG) device is added to support randomness, such as for cryptographic operations. A virtual disk is

configured with the raw image format and connected via the `virtio` interface to ensure optimized I/O performance. Finally, we boot Ubuntu as described in Figure 3.2, resulting in a successful boot of the unmodified Ubuntu distribution.

Extra flags in Qemu to run a real Linux distribution with Miralis
`-device virtio-net-device,netdev=eth0`
`-netdev user,id=eth0`
`-device virtio-rng-pci`
`-drive file=misc/ubuntu-miralis.img,format=raw,if=virtio`

Table 3.4: Qemu arguments for running a real Linux distribution with Miralis

We extend the runner by introducing a new artifact type called *disk*. This artifact enables the use of virtual disks, supporting multiple Linux distributions. Linux distributions are commonly packaged as disk images, which are self-contained snapshots of an entire filesystem. These images include the operating system kernel, user-space utilities, libraries, and pre-configured settings, making them ready to boot and run without additional setup. When a user adds the *–distribution* flag, the runner automatically downloads and extracts the appropriate image for the specified distribution and installs it in the filesystem. Current options include *Ubuntu*, *OpenSUSE*, *Fedora*, and *Debian*.

3.13 Example with Debian on the VisionFive 2 board

Next, we demonstrate that Miralis can run a Linux distribution on the VisionFive2 board. The VisionFive2 board comes with a pre-installed Debian distribution [21]. Following the instruction manual [98], the recommended process involves flashing an external drive using a software called Etcher [4], and booting it with U-Boot SPL, Opensbi, and U-Boot. We successfully launch Miralis along with the corresponding Linux distribution, displaying the output on the screen shown in figure 3.3 and validated the system's stability by conducting a series of rigorous experiments in section 7.

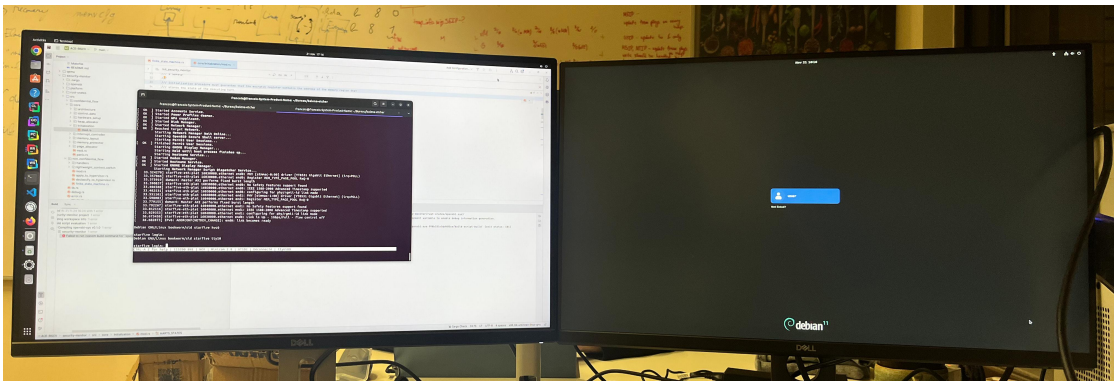

Figure 3.3: Example of Debian running with Miralis on the VisionFive2 board. (Left screen shows the UART output, while the right screen displays the board's output)

4 Running Miralis on the Premier P550 board

Miralis currently supports the Qemu / Spike as well as the VisionFive2 board. Compared to the VisionfFive2 board there are some major differences with the Premier P550 board. While the Premier P550 board and the VisionFive 2 are both RISC-V development boards they target different use cases. The Premier P550 is a high-performance platform built around SiFive's P550 cores, while the VisionFive 2 is a budget-friendly board designed primarily for hobbyists and early adopters of RISC-V. In this section we explore the port of Miralis to the Premier P550 board.

4.1 Understanding the Premier P550 board boot process and how to flash the image

The first part implies understanding the boot process of the board and how to run it. There are multiple documents that are available on the official page such as the software manual, the image manual, the microcontroller manual and the datasheet of the processor [93]. The software manual and image procedure contains all the information required to build and upload the image on the Premier P550 board.

Building the image: The bootchain image for the Premier P550 board consists of the DDR firmware second bootloader, U-Boot, and Opensbi. The image can be built using the Yocto build system or through manual compilation. Using Yocto, the process involves cloning the Freedom U-SDK repository and executing a sequence of BitBake commands to clean and build the bootchain. Alternatively, a manual approach requires obtaining the necessary binaries and source code, including the DDR and second boot firmware the meta-sifive repository, and the U-Boot and Opensbi sources. Patches from meta-sifive must be applied before compiling U-Boot and Opensbi, with the latter using the U-Boot binary as the firmware payload. Finally, the *nsign* tool is used to generate the bootchain image by integrating all components. This approach builds a bootable system for the Premier P550 board. We built the original bootchain using the second method. More details can be found at: *https://github.com/epfl-dcsl/premierp550-doc*

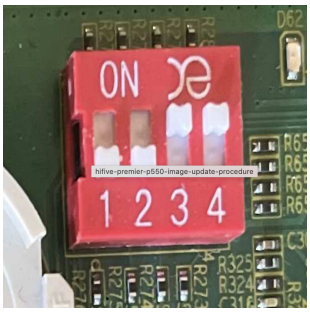

Figure 4.1: Swtiches to boot over USB, borrowed from the image update manual [94]

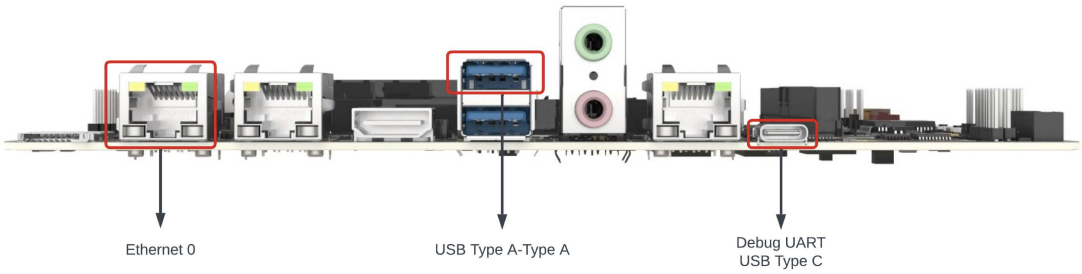

Figure 4.2: Wires to plug on the board to boot over USB, borrowed from the image update manual [94]

Flashing the image on the board: To update the bootchain image on the Premier P550 board, the process begins with powering down the board and make sure that the debug UART is not occupied by any other program on the Host PC. Next, the boot select DIP switch must be set to `0011` to enable USB booting. The USB-c port must be connected to the computer and is a debug UART (`/dev/ttyUSB2`), while the USB-A must also be connected to the board. Upon powering on the board, it enumerates as a USB drive. The next part is to mount the external driver, copy the bootchain image in the mounter folder and unmont it. It takes around one minute to transfer and boot the image properly. If everything works as expected, the bord should print the execution of Opensbi on the board.

4.2 Modifying Miralis for the Premier P550 board

A new board implies a new platform in the Miralis codebase. So we start by adding a platform named Premier P550 board. There are multiple points that depends on the platform such as the address space, the UART, the CLINT driver, the number of available PMP registers. Similarly as for the VisionFive2 board, we place Miralis at the address *0x80080000* such that we don't have to modify the original bootchain locateed at *0x80000000*.

The next steps are the UART and CLINT drivers. Miralis requires for each platform the implementation of two essential drivers, namely the UART driver and the CLINT driver. Documen-

Algorithm 2 Handler redirecting trap to S-mode

1: $next\_is\_virt$ ← (extensions.has_h_extension **AND** prev_is_virt **AND** is_trap(trap_info.mcause))
2: $mstatus \leftarrow mstatus$ **AND NOT** MPV_FILTER
3: **if** $next\_is\_virt$ **then**
4: $mstatus \leftarrow mstatus$ **OR** MPV_FILTER
5: **end if**
6: **if** extensions.has_h_extension **AND NOT** next_is_virt **then**
7: $hstatus \leftarrow$ read_csr(Hstatus)
8: **if** prev_is_virt **then**
9: $hstatus \leftarrow hstatus$ **AND NOT** $SPVP\_FILTER$
10: **if** $prev\_mode = S$ **then**
11: $hstatus \leftarrow hstatus$ **OR** $SPVP\_FILTER$
12: **end if**
13: $hstatus \leftarrow hstatus$ **AND NOT** SPV_FILTER
14: **if** prev_is_virt **then**
15: $hstatus \leftarrow hstatus$ **OR** SPV_FILTER
16: **end if**
17: $hstatus \leftarrow hstatus$ **AND NOT** GVA_FILTER
18: **if** trap_info.gva is true **then**
19: $hstatus \leftarrow hstatus$ **OR** GVA_FILTER
20: **end if**
21: write_csr(Hstatus, $hstatus$)
22: write_csr(Htval, $trap\_info.mtval2$)
23: write_csr(Htinst, $trap\_info.mtinst$)
24: **end if**
25: **end if**
26: **if** NOT $next\_is\_virt$ **then**
27: write_csr(Stval, $trap\_info.mtval$)
28: write_csr(Sepc, $trap\_info.mepc$)
29: write_csr(Scause, $trap\_info.mcause$)
30: $pc \leftarrow$ read_csr(Stvec)
31: $mstatus \leftarrow mstatus$ **AND NOT** MPP_FILTER
32: $mstatus \leftarrow mstatus$ **OR** $(S \ll MPP\_OFFSET)$
33: $mstatus \leftarrow mstatus$ **AND NOT** SPP_FILTER
34: **if** $prev\_mode = S$ **then**
35: $mstatus \leftarrow mstatus$ **OR** $(1 \ll SPP\_OFFSET)$
36: **end if**
37: $mstatus \leftarrow mstatus$ **AND NOT** $SPIE\_FILTER$
38: **if** $(mstatus$ **AND** $SIE\_FILTER) \neq 0$ **then**
39: $mstatus \leftarrow mstatus$ **OR** $SPIE\_FILTER$
40: **end if**
41: $mstatus \leftarrow mstatus$ **AND NOT** SIE_FILTER
42: **else**
43: *Same logic as the if statement with vs registers (except for lines 31 and 32)*
44: **end if**
45: write_csr(Mstatus, $mstatus$)

tation about the two drivers can be found in the EIC7700DX datasheet [93] and the Opensbi patches [92] also gives valuable information anout the triver.

Uart driver: On the firmware we could rely on previous boot steps (such as U-boot SPL) to run the initialization of the UART driver. Unfortunately we can't benefit from this initialization on the Premier P550 board and we have to write the initialization code in the driver as well. Writing the initalization code required a port from the Opensbi code into rust. There are a few differences with the Qemu uart driver such as the base address represented by *EIC770X_UART0_ADDR* at address `0x50900000`. The read and write logic are similar to the VisionFive2 board up to the base address and the stride between the various registers is 32 bits again.

Clint driver: Upon inspection of the EIC7700X datasheet [93], the clint driver is similar to our previous platforms since we have the same memory map. Therefore there is no need to rewrite a part of the clint driver for this new platform and we can use it out of the box.

4.3 Missing features in Miralis to boot the Premier P550 board

A few features were necessary to run Miralis on the Premier P550 board. Surprisingly, read page faults, store page faults and instructions page faults trap to the firmware and are redirected to the payload, which seems to be a bug. Nevertheless, we introduce support to redirect traps from the payload to the payload directly in Miralis. Our implementation is a Rust port of the Opensbi implementation. Similar to Opensbi, this redirection handler also handles the case where the mode comes from the VS-mode.

4.4 Running Miralis on the Premier P550 board

To run Miralis on the board, we need to modify the Nsign image we previously used to build and add the Miralis entry. After that we build the new image and flash it according to section4.1. After this, we boot the board and see that the Premier P550 board Figure 4.3 shows the output of Miralis followed by Opensbi, Uboot and ultimately the Linux kernel with the Ubuntu distribution. Figure 4.3 shows Ubuntu running with Miralis.

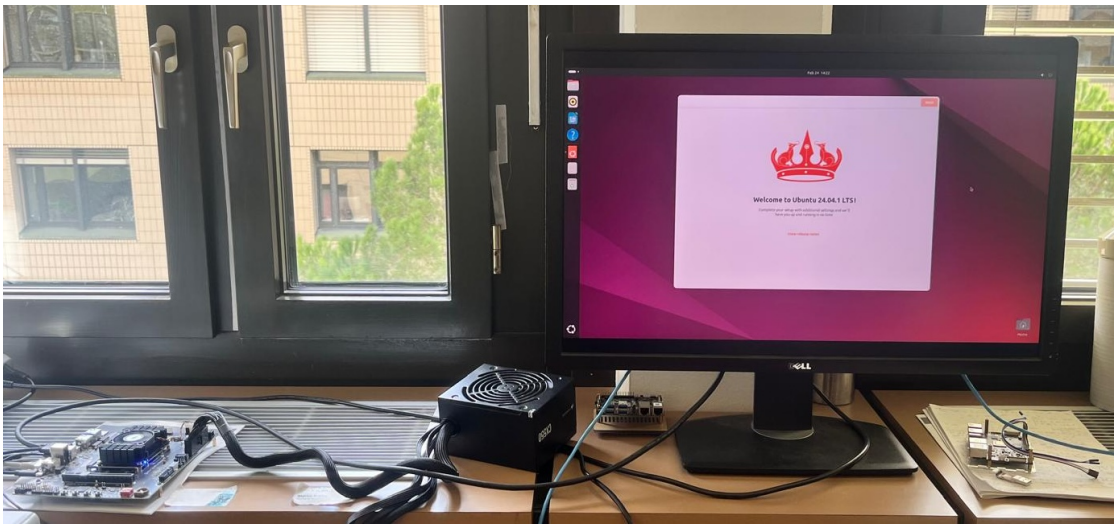

Figure 4.3: Premier P550 board running Ubuntu on top of Miralis

5 Securities policies

Miralis virtualizes the firmware. Running it in a virtualized environment is the first necessary step toward its removal from the trusted computing base (TCB). The second step is to *isolate* components from the firmware. This chapter introduces the concept of *security policies* and demonstrates how various components can be isolated from the firmware. Security policies enforce a set of invariants that protect specific components from the firmware thus isolating the firmware from the TCB.

In this thesis, we explore the isolation of two components. The first policy, the Strict Protect Payload Policy, enforces strict isolation between the Virtual firmware and the S-mode operating system. The Ace Policy aims to protect confidential virtual machines from potential threats posed by the firmware. This policy is more exploratory, as it leverages the recent COVE specification [20] and the ACE security monitor.

Miralis also supports confidential enclaves operating under a third policy: Keystone. However, this policy falls outside the scope of this work.

Policies act as a mechanism that intercepts critical state transitions within the system. In a way, they function similarly to a *trusted* man-in-the-middle (MITM) attack. An man-in-the-middle attack occurs when an attacker secretly intercepts and potentially alters communication between two parties who believe they are directly communicating. This allows the attacker to steal sensitive data, inject malicious content, or impersonate one of the parties. Security policies operate in a similar manner, but with a crucial difference: instead of maliciously altering communications, they transparently conceal confidential information from the firmware (without the firmware being aware) while being trusted by the rest of the system.

Table 5.1 presents an overview of the state transitions intercepted by Miralis' security policies. If a policy does not intercept a transition, Miralis forwards the unmodified exception to the firmware.

At the end of this chapter, we introduce a final policy that differs from the others and can be combined with any security policy. We call this the Offload policy . A key observation in

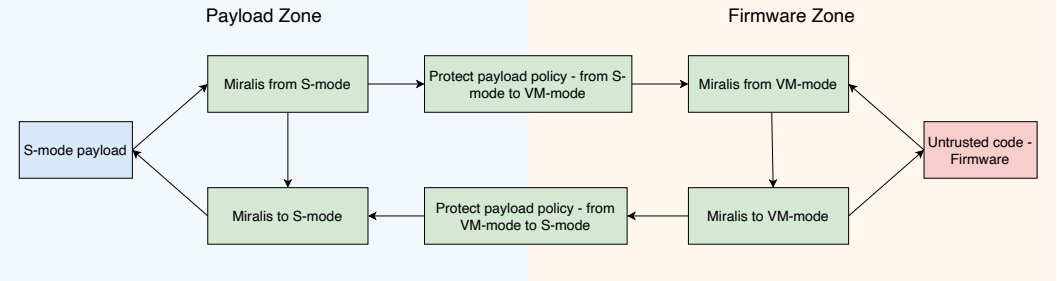

Figure 5.1: State in Miralis

Interception	Description
Exception from firmware	There is an exception from the firmware
Exception from payload	There is an exception from the payload
Transition from firmware to payload	Miralis leaves the firmware
Transition from payload to firmware	Miralis enters the firmware
Machine software interrupt	Miralis receives a machine software interrupt

Table 5.1: State transitions intercepted by the policy

the evaluation of Miralis (Section 7) is that the RISC-V firmware plays a third role beyond hardware management and security enforcement: it emulates software features that the RISC-V hardware does not yet support. Some operations trigger frequent exceptions to the firmware leading to significant overhead—especially in I/O-intensive workloads—where the cost of a round trip to the firmware is unacceptable due to performance degradation. To address this, we implement a policy that directly handles these operations within Miralis, mitigating performance overhead.

5.1 Default policy

The *default policy* is applied *de facto* when no policy is running on Miralis. Its implementation is trivial, as it ignores all intercepted events.

5.2 Strict Protect Payload Policy

The Strict Protect Payload Policy enforces strict isolation between the S-mode operating system payload and the virtual firmware monitor. This policy ensures both integrity and confidentiality for the S-mode operating system operating system.

First, we discuss the threat model in Section 5.2.1. Sections 5.2.2 and 5.2.2 respectively detail how we protect memory and registers. The final two sections 5.2.7 and 5.2.8 explore multicore

integration and the challenges encountered on real hardware.

5.2.1 Threat model

In this scenario, we assume a software-level adversary capable of controlling the untrusted firmware known as the firmware. The adversary's goal is either to extract confidential information from the operating system or its child processes and/or to manipulate the victim to alter its execution flow.

The firmware can be any binary blob, but we assume it cannot access device drivers except for those protected by the CLINT driver. This threat model excludes denial-of-service attacks, physical attacks, and side-channel attacks.

5.2.2 Memory protection

The Strict Protect Payload Policy protects the payload's memory by employing a PMP slot specifically to isolate the S-mode operating system operating system payload. When transitioning from the firmware to the payload (*from_firmware_to_payload*) or from the payload to the firmware (*from_payload_to_firmware*), the policy activates and deactivates the PMP accordingly.

To further validate our hypothesis, we explicitly filtered the SBI calls received during the Linux kernel boot process and observed no calls sharing an address space. A minor exception occurs when booting Ubuntu, which we discuss in Section 5.2.8.

General-purpose register masking

In addition to memory protection, the Strict Protect Payload Policy hides the registers of the S-mode operating system payload during context switches, as required for by the confidentiality property. The firmware could otherwise exploit register contents to leak confidential data. This measure prevents such leaks.

During the logical transitions *from_firmware_to_payload* and *from_payload_to_firmware*, the policy clears and restores the general-purpose registers, masking them from the firmware.

However, hiding and restoring all registers is not always the desired behavior, as it enforces a strict isolation between the two components. For example, the firmware and the payload communicate via the SBI interface [38]. This necessitates a more flexible approach: rather than hiding all registers, we selectively allow registers *a0–a7* to pass and return values in *a0–a1*, as required by the SBI interface.

The Strict Protect Payload Policy enforces even stricter isolation in the presence of an ECALL. Each SBI call is uniquely identified by a tuple *(eid, fid)* and has a fixed number of arguments.

To enforce this rule, we developed a script that automatically parses the SBI interface, retrieves the exact number of arguments for each *(eid, fid)* pair, and generates a Rust function to filter out the other registers.

Parsing the SBI interface required approximately 100 lines of Python code and did not introduce unintended side effects. On the firmware some registers must also be forwarded in cases of illegal instruction emulation. We discuss this edge case in Section 5.2.5.

SBI Call	Description
4. System Boot Interface	
4.1	Get SBI specification version (FID #0)
4.2	Get SBI implementation ID (FID #1)
4.3	Get SBI implementation version (FID #2)
4.4	Probe SBI extension (FID #3)
4.5	Get machine vendor ID (FID #4)
4.6	Get machine architecture ID (FID #5)
4.7	Get machine implementation ID (FID #6)
6. Timer	
6.1	Set Timer (FID #0)
8. Inter-Processor Interrupt	
8.1	Remote FENCE.I (FID #0)
11. Performance Monitoring Unit	
11.5	Get number of counters (FID #0)
11.6	Get details of a counter (FID #1)
11.7	Find and configure a matching counter (FID #2)
11.9	Stop a set of counters (FID #4)
11.12	Set PMU snapshot shared memory (FID #7)

Table 5.2: SBI calls intercepted during the boot of the Linux kernel

5.2.3 CSR register masking

Another category of registers that leak information is the control and status registers (CSRs). Although RISC-V provides many registers, we only need to manage a subset of them, as we are isolating supervisor CSRs from the firmware. *Automatic ISA analysis for Secure Context Switching* [53] is a project from *IBM Research* and *EPFL* that automatically detects security-sensitive registers. It employs static analysis on the official RISC-V Sail specification [83] to identify registers that expose sensitive information. We use the generated list to explicitly hide these registers from the payload.

Register hiding occurs in two phases. In the first phase, while the firmware prepares the payload, it has access to the entire CSR register space. Upon the first jump to the payload, the Strict Protect Payload Policy categorizes the registers into two groups. The first group

consists of M-mode CSR registers, which remain under the firmware's control. The second group consists of S-mode operating system registers, which are managed by the payload. When transitioning from the payload back to the firmware these registers are cleared and become inaccessible to the firmware. Any attempt to modify them is ignored. The *MIE*, MIP, and MSTATUS registers are exceptions, as they are shared between both modes. The Strict Protect Payload Policy resolves this by managing access at the *bit granularity* level: subfields affecting supervisor-mode logic remain under supervisor control, while the rest are handled by the virtual firmware. Miralis clears and restores the payload values using bitwise operations. Since CSR registers are managed on a per-core basis, no synchronization is required between different cores, unlike in shared-memory models.

5.2.4 Emulation of misaligned loads and stores

Simple emulators such as Qemu and Spike automatically emulate misaligned loads and stores. Unfortunately, this property does not hold for a general RISC-V platform, where misaligned memory accesses trigger a trap in the VisionFive2 board. In the absence of a dedicated policy, Opensbi handles these traps by emulating the misaligned access and resuming execution. Delegating this task to untrusted firmware however, contradicts the strict isolation requirements. Therefore, we perform the emulation directly in Miralis. While this approach is not strictly necessary—emulation could still be delegated to the firmware via a bounce buffer allowing controlled memory access—it ensures stronger isolation.

5.2.5 Handling of illegal instructions

The VisionFive2 board does not support the delegation of the time register to supervisor and user modes. Similar to misaligned loads and stores, certain values must pass through the firmware. However, in this case, the filtering process is dynamic. The instructions CSRRW, CSRRS, CSRRC involve both an rs1 and an rd register, whereas their immediate counterparts, CSRRWi, CSRRSi, CSRRCi, only involve an rd register. In the former case, both rs1 and rd values are forwarded to and retrieved from the firmware. The latter case follows a similar approach but only involves a single value.

5.2.6 Early jump attestation

Before executing the first jump into the S-mode operating system, it is essential to ensure that the firmware has loaded the payload correctly and that the system is in a known, valid state. For instance, upon transitioning to S-mode operating system, the program counter must point to the payload's entry point. We verify the security integrity of the S-mode operating system payload using cryptographic attestation from the Strict Protect Payload Policy, ensuring that the firmware is genuine and unaltered, thereby preventing any loss of integrity.

5.2.7 Integration of multiple Cores

Transitioning to a multicore setup with Strict Protect Payload Policy introduces additional synchronization challenges. Specifically, when execution switches to the payload, Miralis performs attestation before transferring control. However, this naive approach presents a security risk: other cores could remain in the firmware and access the payload's memory, as RISC-V's Physical Memory Protection (PMP) operates on a per-hart basis. Consequently, a malicious firmware could reserve a core indefinitely, prevent it from entering supervisor mode (S-mode operating system), and access protected memory at any point during execution.

To mitigate this risk, Miralis must synchronize execution and lock memory access as soon as the first core enters the payload. This can be achieved using the *Inter-Processor Interrupt (IPI)* mechanism in RISC-V.

Upon receiving the IPI, the Strict Protect Payload Policy immediately locks down memory access, preventing any remaining cores in the firmware from accessing the payload. Once this security measure is enforced, normal execution resumes in the virtual firmware allowing each core to continue its tasks securely.

5.2.8 Testing the policy

We first test the protected payload policy by creating a sandboxed VM-mode firmware and an S-mode operating system operating system payload environment that intentionally attempt to communicate with each other, read unauthorized memory locations, and alter the state of specific registers before jumping back to the S-mode operating system operating system payload. We then verify that the values visible in each system match the invariants defined in the Strict Protect Payload Policy. Finally, we integrate this firmware into the CI/CD pipeline to ensure that every future commit complies with the defined requirements.

Next, we launch the Strict Protect Payload Policy with the Linux kernel using the boot procedure illustrated in figure 3.2 on Qemu and Spike. The Linux kernel is compiled with the default RISC-V configuration and loaded at address *0x80400000*. Preventing the firmware from accessing the payload's memory did not cause any undesired side effects when running Debian, Fedora, and Ubuntu (in recovery mode). However, the policy failed to function correctly when booting the full Ubuntu distribution on Qemu and Spike.

At system boot, the firmware and the S-mode operating system payload engage in a communication process formally defined by the SBI . Ubuntu requests the firmware to print characters on the screen using the *debug console write* function. In this ecall, the payload must provide an address range specifying the buffer to print. We resolve this issue by returning the error code *-4*, corresponding to *SBI_ERR_DENIED*. This is not a fundamental limitation of our approach, as the call can be forwarded using a *bounce buffer*.

On the VisionFive2 board board, the policy initially failed to boot for two reasons. First, Qemu

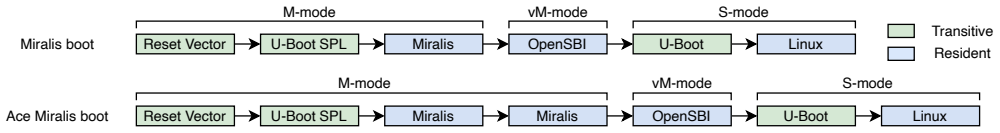

Figure 5.2: Comparison of the Miralis and Ace Miralis boot flow

and Spike (by default) emulate misaligned loads and stores, whereas the VisionFive2 board board does not, as discussed in Section 5.2.4. Consequently, explicit handling of misaligned memory accesses was necessary for the VisionFive2 board. Second, certain illegal instructions from the payload required specific registers to be passed to the firmware for correct emulation. After addressing these two issues, the Strict Protect Payload Policy successfully booted on the VisionFive2 board.

5.3 Ace Policy

The Ace Policy enforces isolation between confidential virtual machines and the firmware. In addition to the default confidentiality guarantees that confidential VMs has with respect to the operating system, the policy ensures both integrity and confidentiality for CVMs with respect to the firmware.

5.3.1 ACE security monitor

The Miralis project explores the isolation of two security environments from the firmware: confidential virtual machines and enclaves. Enclaves are protected by the Keystone policy, which is outside the scope of this work, as it was implemented by another student as part of a semester project [55]. This section focuses on isolating the firmware from confidential VMs. At a high level, we colocate the Ace security monitor with Miralis. Ace distinguishes itself from other existing security monitors by operating based on the new RISC-V Cove specification [85, 20], a standard that we anticipate will gain significant adoption in the future.

ACE defines a set of axioms that describe the necessary hardware properties required to run the security monitor. The authors refer to this set of axioms as the *canonical architecture*. These axioms emphasize, for example, the need for multiple execution privilege levels (similar to the work of Popek and Goldberg), the protection of initial boot code from unauthorized modifications, and mechanisms for isolating confidential memory. One of the key axioms asserts that only software running at the highest privilege level can reconfigure security-critical hardware components.

The Ace security monitor starts with an initialization phase, during which it ensures that the security monitor controls the entire computing environment, guaranteeing that no unauthorized software can modify its code or access critical security components, such as the endorsement seed used for attestation. The monitor's runtime operation is governed by a

finite state machine (FSM), which manages state transitions between non-confidential and confidential domains. This structure allows the monitor to enforce strict access controls and maintain well-defined security invariants during interactions between different security domains.

Similar to Miralis, Ace leverages the memory safety guarantees provided by Rust to reduce memory-related vulnerabilities. The security monitor utilizes several optional RISC-V extensions, in particular, atomic extensions for synchronization and the hypervisor extension for virtual machines. These features are required by the axioms defined in the theoretical model and enable Ace to effectively enforce its security guarantees.

A core component of the Ace security monitor is the memory tracker, which allocates confidential memory at page-size granularity. The memory tracker ensures that two disjoint security domains, such as different CVMs, cannot share the same physical memory region. This is achieved through a token-based ownership model, similar to Rust's ownership system, which simplifies reasoning about memory allocation and access control.

5.3.2 Threat model

The canonical architecture (interrupt controller, immutable boot code, endorsement seed, atomic instructions, physical memory isolation, execution privilege separation, random number generator) assumes a software-level adversary capable of controlling all untrusted software, including the hypervisor, operating system, firmware user-space software, virtual machines (VMs), and some confidential VMs (CVMs), except for the victim CVM. The attacker's objectives include: (1) extracting confidential data from the victim CVM, (2) manipulating the victim CVM to process adversary-supplied data, (3) altering the execution flow of the victim CVM, and (4) impersonating the victim CVM to its user or owner. The attacker can arbitrarily start, stop, or interrupt the victim CVM at any point and inject data via virtual I/O devices, registers, or shared memory buffers. Additionally, the attacker has arbitrary control over the virtual firmware monitor and can interact with peripheral devices that are not assigned to any confidential VM. Similar to Ace [75], we explicitly exclude denial-of-service attacks, physical attacks, and side-channel attacks.[I]

5.3.3 Colocating Ace with Miralis

Despite Ace offering strong, formally verified security guarantees between the untrusted hypervisor and confidential virtual machines, under the modified version of the threat model 5.3.2, the invariants are trivially broken since the firmware is at the root of trust and operates with the highest privilege because Ace colocates the Opensbi firmware in the trusted computing base. In the rest of this chapter, we demonstrate how Opensbi can be removed from the TCB

[I] Note for plagiarism: This threat model is a slightly modified version of the original threat model for the Ace security monitor [75] and is not my work.

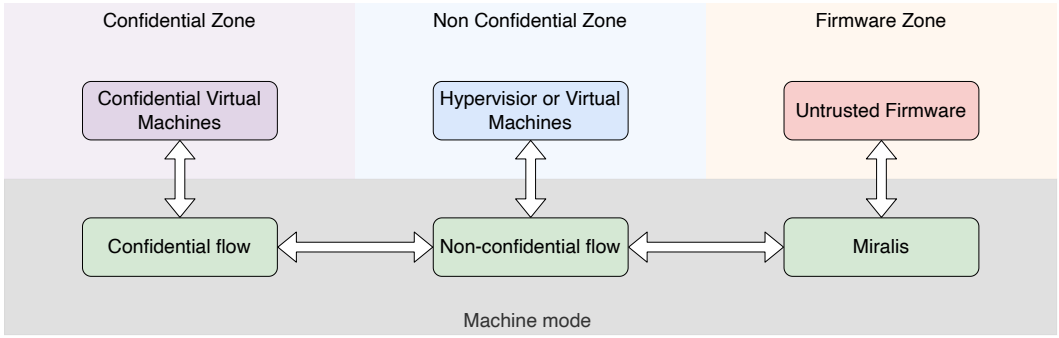

Figure 5.3: Logical transitions between Ace and Miralis

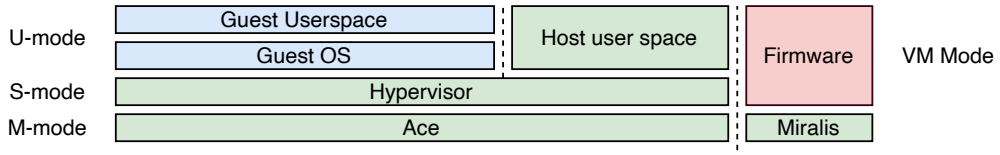

Figure 5.4: ACE in colocation with Miralis

and relocated to VM-mode.

Our system disrupts the Ace -OpenSBI colocation and introduces Miralis as a layer of indirection between Ace and Opensbi. Opensbi runs in VM-mode on top of Miralis. The boot process begins with Miralis taking full control of the system. At some point during the boot process, it modifies the *Flattened Device Tree* as required before jumping to Ace 's initialization phase. Once Ace completes its initialization, control returns to Miralis, which then jumps to virtual firmware mode. Figure 5.2 illustrates the boot procedure.

Runtime state transitions between Miralis and Ace

During the rest of the execution, we introduce new states and logical transitions in the Ace -based finite state machine. There are two main sets of states: the *confidential flow* and the *non-confidential flow*. Interrupts delegated to Opensbi always occur in the *non-confidential flow*. Therefore, we route all transitions from Opensbi to Miralis, leading to the finite state machine shown in Figure 5.3.

Having two security monitors in the design introduces new challenges. Both Miralis and Ace are M-mode security monitors, which require full control over the hardware. According to the pigeonhole principle, there is no room for both at the same time, as there is only one underlying hardware. There are two solutions to resolve this issue: we can either merge them into a single component or relax the original requirements and design a different architecture. We opt for the second approach and implement a *colocation* between the two components, based on the following core principle: If the S-mode operating system is running (hypervi-

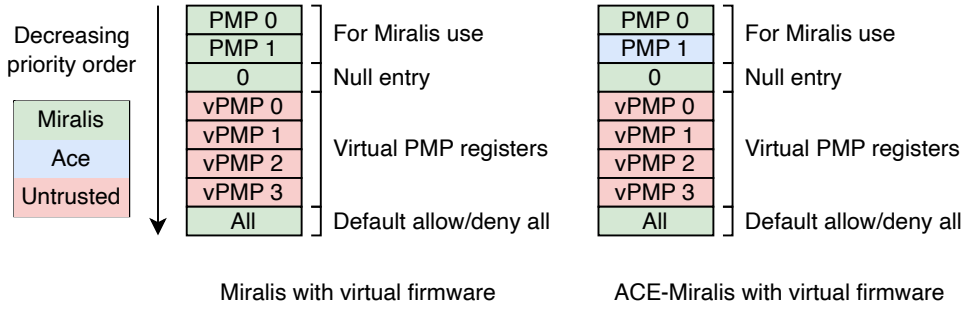

Figure 5.5: Comparison of PMP Layout between Ace and Miralis

sor, VMs, or CVMs), the underlying security monitor is Ace . If the VM-mode firmware is running, the underlying security monitor is Miralis. When transitioning between the two security monitors, the first security monitor delegates full control to the second one. This transition specifically involves changing the *mscratch* register (while preserving the address of the other mscratch location) and the address of the *mtvec* register. We refer to this transition as a *Security Monitor Context Switch*. State transitions between these zones are managed directly by the *Policy module*. Transitions between Ace and Miralis are initiated by the functions *Policy.from_non_confidential_to_confidential* and *Policy.from_confidential_to_non_confidential*. During these transitions, the system transfers the ownership state, installs the appropriate trap handler, and restores the previous state. The advantage of the second approach over the first is simplicity—it only requires writing a few wrappers between the two security monitors.

This approach raises concerns about memory protection, as we must ensure that both security monitors are protected at all times. Otherwise, the firmware could modify Ace 's contents, set the system to a state where Ace jumps to the firmware in M-mode while Miralis is running, and, upon transitioning to Ace , grant full privileges to the firmware. This would break the threat model defined in 5.3.2. Similarly, a virtual machine could modify Miralis, instruct Miralis to jump to a defined location in the virtual machine in M-mode, and then leak the contents of the confidential virtual machines. Therefore, we do not fully transfer the PMP registers. Instead, each security monitor is assigned a disjoint subset of the available PMP registers at compile time, as shown in Figure 5.5. Each component is responsible for protecting itself from the rest of the system. Miralis determines at compile time how many PMP entries it requires, and these are allocated and made available to Ace . Although both security monitors have the capability to overwrite each other's PMP settings, this is not an issue since both components are part of the trusted computing base.

Building the boot image and the Linux kernel

In this section, we describe how to build the binary required to run the Ace security monitor with Miralis. The implementation is based on the COVE specification, which requires a patch

to the Linux kernel. Therefore, we use the prebuilt Linux kernel from the Ace repository by running the *make hypervisor* command. This command applies the *cove.patch* [45] and *ace.patch* [44] and then compiles the Linux kernel along with the initial filesystem, which includes another instance of Qemu as well as a Linux kernel to run the confidential VM.

We build the entire Ace toolchain using the command *MAKEFLAGS="–silent -j4" make*, as mentioned in the original README. The next step is to extract the Linux kernel and compile an unmodified version of Opensbi. The Opensbi patches include a function call to the Ace initialization code and shift the starting PMP. Miralis addresses the first issue by implementing theAce initialization code directly. Additionally, the PMP does not need to be shifted in Miralis, as Opensbi's PMP settings are virtualized. Therefore, we do not use their patched firmware but rather the original Opensbi with the patched Linux kernel as the embedded payload.

Running the Ace Policy

Currently, theAce security monitor is an ongoing project with no support for real hardware. We have been in contact with the team at IBM Zürich, and as of the time this thesis was written, the Ace security monitor does not yet support booting on real hardware. Therefore, we leave this policy at the proof-of-concept stage. The proof of concept works on Qemu with multiple cores. We leave the implementation of this policy on real hardware as future work.

5.4 Offload policy

The evaluation section 7 demonstrates that real workloads suffer from a significant number of exceptions, leading to unacceptable overheads in real applications. In the extreme case of a balanced mix of reads and writes, we observe a 200% overhead in the Redis benchmark. Since Miralis introduces additional overhead through trap and world switches, we developed a kernel driver in the Linux kernel to trace the number of firmware traps and world switches, categorizing them by function in the system. Our analysis shows that almost all firmware activity corresponds to software emulation due to the lack of hardware support for certain features. Four categories of exceptions are predominant:

- Misaligned Loads and Stores: These occur when the base address of an operation is not aligned to a multiple of its size. Miralis emulates these unaligned memory accesses, resulting in performance penalties.
- Time Register Reads: Modern hardware allows the kernel to read time directly through the time register when enabled by the *mcounteren* register. However, on the VisionFive2 board board, this feature is not available. Reading this CSR register triggers an illegal instruction exception, which Opensbi decodes and emulates.
- Supervisor Timer Interrupts: On modern hardware with the SSTC extension, the payload can write directly to the *stimecmp* register to schedule interrupts (e.g., for the kernel scheduler). On older hardware like the VisionFive2 board board, this feature is unavail-

able, and the payload must request the firmware to trigger a timer interrupt via an SBI ecall.
- Remote Fences: Less frequently, the payload requests the firmware to trigger an S-mode operating system IPI to other cores on the board.

Although these exceptions will not appear on future hardware (which will support these features natively), they currently present a substantial obstacle to efficient firmware virtualization. To mitigate the performance overhead, we introduce the offload policy, which offloads misaligned loads and stores, time register reads, and supervisor timer interrupts to the payload. As shown in the evaluation section, offloading these operations eliminates the performance gap between native firmware execution and its virtualized counterpart.

5.4.1 Offload misaligned loads and stores

The Strict Protect Payload Policy already supports misaligned loads and stores. We have ported this logic into the new offload policy. This integration works seamlessly and does not require any modifications.

5.4.2 Offload time register reads

In RISC-V, the time register is a special-purpose supervisor and userspace CSR register that tracks elapsed time since a certain event, typically system startup. It is commonly used for scheduling and performance measurement. The *mcounteren* and *scounteren* registers can delegate access to memory-mapped mtime registers for Supervisor and Userspace modes. Unfortunately, this delegation is not supported on all RISC-V platforms. When it is not available, the payload must rely on the firmware's trap handler to emulate the illegal instruction triggered by reading the time register. In Miralis, this involves a full round trip through the firmware introducing a significant overhead of several thousand cycles. To address this issue, we bypass the overhead by returning the timer directly from Miralis.

5.4.3 Offload interrupt management to Miralis

The third offload, interrupt management, is the most complex of the four. Platforms that lack the SSTC extension [102] must send *ECALLs* to the firmware via the SBI interface [38], and the firmware injects interrupts when the designated period has elapsed. Offloading this logic to Miralis requires demultiplexing the hardware timer. The Miralis virtual CLINT contains two buffers to track when the next firmware and payload interrupts should be triggered. It writes the minimum of the two values to the physical CLINT. Upon receiving an interrupt, Miralis injects the corresponding interrupt by setting the appropriate bit in the virtual context, then adjusts the physical CLINT by writing the minimum value and setting the other value to infinite.

Algorithm 3 Multiplexing timer interrupts

```
1: if origin = ClintTimer::Payload then
2:     next_timestamps[current_hart].counter_payload ← value
3:     ctx.csr.mip ← ctx.csr.mip AND ¬ STIE_FILTER
4: else
5:     next_timestamps[current_hart].counter_firmware ← value
6:     ctx.csr.mip ← ctx.csr.mip AND ¬ MTIE_FILTER
7: end if
8: is_interrupt_ready ← (mtime ≥ value)
9: if is_interrupt_ready then
10:    if origin = ClintTimer::Payload then
11:        ctx.csr.mip ← ctx.csr.mip OR STIE_FILTER
12:    else
13:        ctx.csr.mip ← ctx.csr.mip OR MTIE_FILTER
14:    end if
15: else
16:    mtimecmp_firmware ← next_timestamps[current_hart].counter_firmware
17:    mtimecmp_payload ← next_timestamps[current_hart].counter_payload
18:    driver.write_mtimecmp(hart, min(mtimecmp_firmware, mtimecmp_payload))
19: end if
20: if origin = ClintTimer::Payload then
21:    propagate_payload_interrupt_physically(ctx)
22: end if
```

One important detail is that Miralis transfers the registers to the virtual context and hardware when switching from the firmware to the payload. When injecting an interrupt into the payload while trapping from it, we must explicitly write the *mip* register in hardware. If we neglect this step, the interrupt will only trigger after a world switch. This dependency on the interrupt could lead to a potential deadlock in the system.

Algorithm 4 Demultiplexing timer interrupts

clint ← `Plat::get_clint().lock()`
current_timestamp ← clint.read_mtimecmp(current_hart)
if *current_timestamp* ≥ *next_timestamps*[current_hart].counter_firmware **then**
    next_timestamps[current_hart].counter_firmware ← ∞
    ctx.csr.mip ← ctx.csr.mip **OR** `MTIE_FILTER` ▷ Inject virtual interrupt to firmware
end if
if *current_timestamp* ≥ *next_timestamps*[current_hart].counter_payload **then**
    next_timestamps[current_hart].counter_payload ← ∞
    ctx.csr.mip ← ctx.csr.mip **OR** `STIE_FILTER` ▷ Inject virtual interrupt to payload
end if
new_timestamp_firmware ← *next_timestamps*[current_hart].counter_firmware
new_timestamp_payload ← *next_timestamps*[current_hart].counter_payload
clint.write_mtimecmp(mcurrent_hart, min(*new_timestamp_firmware*, *new_timestamp_payload*))

5.4.4 Superivsor level IPI

The S-mode operating system requests the firmware to broadcast an IPI (Inter-Processor Interrupt) to other cores. To optimize this, we implement direct support in Miralis to perform Supervisor-level IPIs without relying on the firmware. Within the virtual clint, we introduce an additional array of atomic boolean variables, shared across all cores. When the firmware requests an interrupt, we check the mask to determine which core should receive the interrupt. We then mark the relevant cores and trigger the interrupt on the physical CLINT. Upon receiving the interrupt, the other cores check the array and trigger the Supervisor IPI if the corresponding bit is set.

6 Lightweight formal verification of Miralis

*Lightweight Hypervisor Verification: Putting the Hardware Burger on a Diet - Charly Castes, **François Costa**, Thomas Bourgeat, Nate Foster and Ed Bugnion - to appear at HotOS XX*

Over the past few decades, hypervisors have become an integral part of computing systems. They now operate on virtually every cloud server, are standard on consumer laptops and smartphones, and are widely used in IoT and safety-critical devices. However, with this widespread adoption comes an increased responsibility for ensuring the security and reliability of these systems. As virtualization becomes the default for compute infrastructure, hypervisors are expected to meet high standards of dependability.

Overall, the security of a system can only be as good as the security guarantees given by the components in the root of trust. Bugs in software systems are inevitable. Complex software often contains many bugs and unexpected behaviors. Industry guidelines [68] suggests that an average of 15–50 bugs are expected per 1,000 lines of delivered code. Even in the presence of a minimal codebase and defensive programming principles such as Rust to leverage memory safety guarantees [108], the absence of a heap allocator, this don't guarantee *the absence of bugs*. In this section, we show how we can leverage the official sail RISC-V specification and mathematically prove the *absence of bugs* through *formal, machine-checked verification* [29, 42, 88, 106] using *lighweight formal methods*. We release the formal verification process in open-source, which increases transparency. Developers and researchers can independently inspect the process, increasing Rust in the system, similar to Intel's TDX module [23] or ARM's TF-RMM [36].

6.1 Background and related work

Reducing the amount of trusted code and shifting non-security critical code in userspace is the most well-known strategy to decrease the probability of bugs. Various systems are based on this foundational principle, such as OS microkernel [1, 12, 43, 64, 91, 113, 65, 84], isolation kernels [112, 47], microhypervisor [26, 40, 89, 97, 15], type-safe language kernels [8,

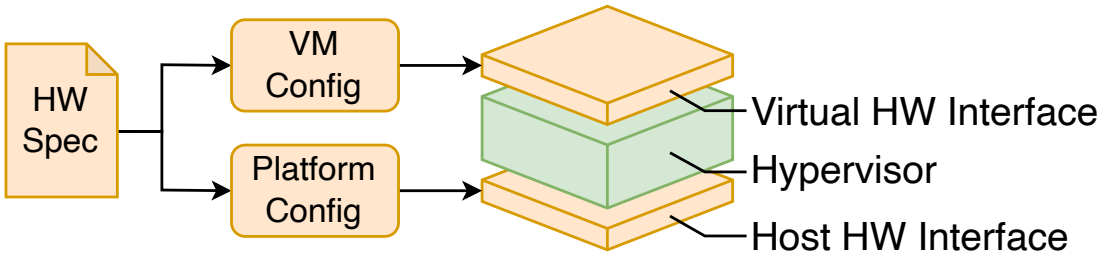

Figure 6.1: The hardware burger

31] and webbroswers [52]. While these architectures help mitigate vulnerabilities, they are not formally verified for correctness. Nevertheless they are a first step towards formal verification, as there is a clear definition of what needs to be verified and make the formal verification computationally feasible.

Operating systems and hypervisor are crucial components and many work addresses the verification problem. seL4 [56] verifies the full functional correctness of a 7500-LOC microkernel C but is a resource-intensive process (20 person-years). mCertiKOS [41] adopts layered abstract specifications to reduce interdependencies, enabling a more tractable verification of a stripped-down kernel. VCC [59] verifies functional correctness for a fixed Hyper-V hypervisor using automated theorem proving but covers only a tiny fraction of the codebase. SeKVM [62] introduces the concept of layered verification to enhence speed. Serval [72] and Komodo [33] relies on hand-written specification and have only a partial support for the full ISA. Nickel [95] proves noninterference with a low proof burden.

Islaris [86] introduces a method to formally verification against authoritative ISA efficieltly and is a framework that employs SMT-based symbolic execution and has been successfully applied to both RISC-V and the more complex ARM ISAs. The üXMHF hypervisor [109, 110] don't verify any functionnal corectness but can verify simpler propersites such memory integrity of multiprocessor. [87, 107, 114] verify the memory management unit subsystem within the scope of an OS kernel. Various work on formal methods also handle the problematic of filesystems [115, 17, 16, 46, 96]. Formal shim verification[52] verifies a sequential browser kernel with just a few hundred lines of code and enforces noninterference between sandboxed components. The Compcert project [60] formally verifies the compilation of a subset of the C programming language.

6.2 Putting the Hardware Hamburger on a Diet: Lightweight Hypervisor Verification...

Despite significant advances in verifying hypervisors for specific use cases, these efforts tend to focus on individual systems and rely on tools tailored to particular scenarios. To extend verification to the vast and diverse range of production-grade hypervisors [13, 105, 67, 5, 22,

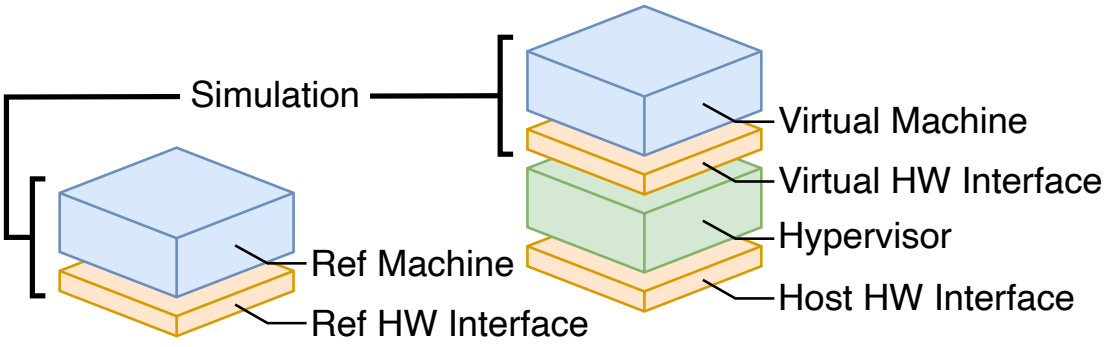

Figure 6.2: Simulation setup

61, 111], there is a pressing need for structured, broadly applicable criteria that guide the application of formal methods.

Instead of targeting a specific implementation, our approach examines the fundamental concept of *correctness* in hypervisors and develops a framework to guide future verification. This methodology is grounded in two key hardware trends: the virtualization capabilities of modern architectures and the growing availability of high-quality, well-maintained hardware specifications. Over the last decade, major architectures have incorporated virtualization features by design, or through extensions that adapt older instruction set architectures (ISAs) like x86-32 to support virtualization.

Building on the foundational work by Popek and Goldberg [78], which provided criteria for assessing ISA virtualizability in the 1970s, we expand their framework to define correctness standards for modern hypervisors. High-level properties, such as isolation and compatibility with existing platforms, emerge from accurately simulating a reference machine. Using the layered structure of hypervisors—conceptually represented as the *hardware burger* in figure 6.1, we derive sufficient conditions for achieving equivalence in both execution modes of contemporary hypervisors. We introduce a new formalization of hypervisor correctness, presenting two key criteria: *faithful emulation* 6.4.2 and *faithful execution* 6.4.3. These criteria are designed to leverage existing hardware specifications, minimizing the need for additional development efforts and significantly simplifying the verification process for hypervisors.

6.3 A simulation problem

Key properties of hypervisors, such as *isolation* and *compatibility*, stem from addressing a simpler simulation problem. This problem involves demonstrating that the virtualized program—commonly referred to as the *virtual machine* (VM)—is effectively a simulation of an independent reference machine. The accompanying figure provides an overview of the host, virtual, and reference machines. A critical insight is that while the host and reference machine are based on the same architecture, their configurations can differ significantly. Examples of

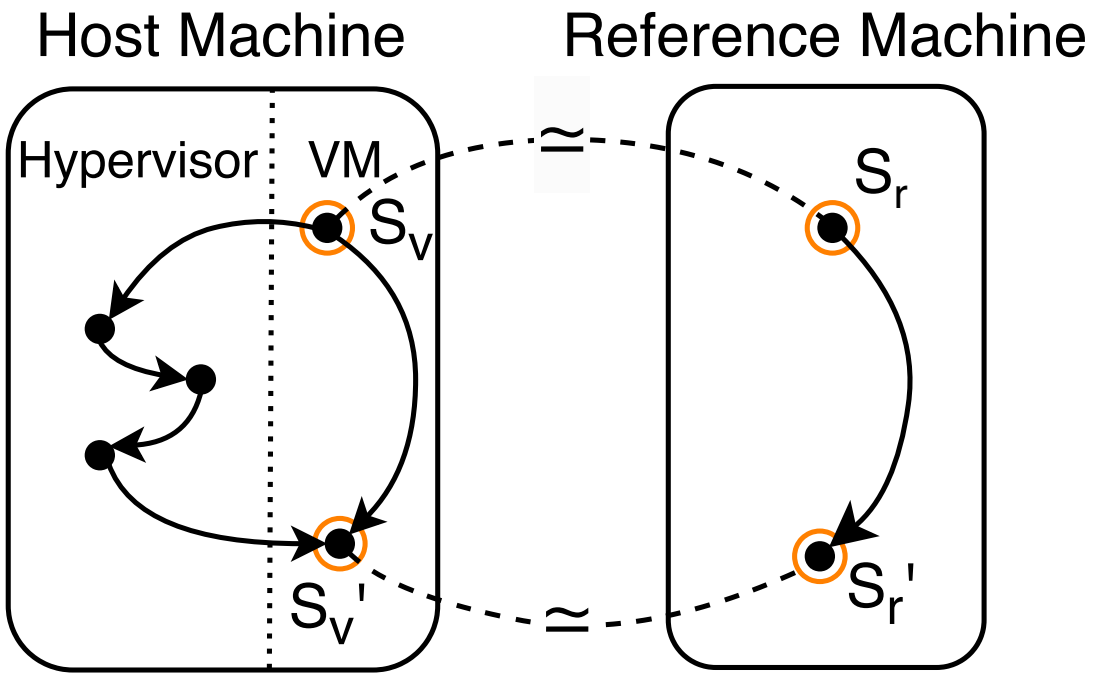

Figure 6.3: Lock steps

such differences include memory size, the number of cores, interrupt identifiers, or hardware extensions. To illustrate, consider the concept of memory isolation. If the reference machine is configured to utilize only a subset of the host memory, proving that the VM correctly simulates the reference machine ensures it cannot access memory outside this subset. Any attempt to do so would result in a fault (e.g., an invalid address error) on the reference machine, and consequently, the same fault would occur on the VM.

Establishing a connection between the behavior of the virtual and reference machines requires detailed models of both. For the reference machine, this is straightforward, as it can be represented using the hardware specification combined with a set of platform parameters (such as memory or cores). Modeling the VM, however, requires an understanding of the hypervisor's behavior.

Modern hypervisors typically rely on the *trap-and-emulate* approach, a concept originally formalized by Popek and Goldberg in their seminal work on virtualization requirements. Under this method, the VM's instructions are executed in one of two modes: *direct execution* or *emulation*. *Direct execution* occurs for instructions that can run natively on the hardware, while *emulation* is used for instructions that must be intercepted and handled by the hypervisor. To validate that the VM correctly simulates the reference machine, we propose the concept of *lock-step execution*. The proof hinges on defining and maintaining the equivalence of "observable states" between the virtual and reference machines at each step of execution.

Observable states are defined based on the Popek and Goldberg theorem, which distinguishes

between privileged and unprivileged states. The privileged state comprises architectural elements that can only be accessed via privileged instructions, which cause the hypervisor to intervene. Conversely, unprivileged state includes all other architectural elements, which the VM can access without interference. By carefully managing how the hypervisor emulates privileged instructions, specific parts of the privileged state can be exposed to the VM when necessary, for example, by mapping virtual privileged registers into general-purpose registers.

Definition 6.3.1 (Lock-step execution)**.** The virtual and reference machine execute in *lock-step* if their initial observable states are equal, and if after each instruction executing on the virtual and reference machine their observable states remain equal.

The virtual and reference machines are said to execute in *lock-step* if they begin with identical observable states and if, after each executed instruction, their observable states remain equivalent. To prove lock-step execution, we analyze two cases. If the instruction being executed is privileged, it triggers a trap to the hypervisor, which emulates it in software. Notably, this entire emulation process is treated as a single instruction from the VM's perspective. On the other hand, unprivileged instructions are executed directly on the hardware. The figure on lock-step execution illustrates this process, showing how privileged and unprivileged instructions are handled differently to maintain consistency between the virtual and reference machines.

6.4 Correctness criteria

The challenge of verifying complex high-level properties, such as memory isolation, is simplified by reducing it to the more straightforward task of ensuring lock-step execution. In this section, we define two key criteria — *faithful emulation* and *faithful execution* — which provide guarantees of lock-step execution for the emulation of privileged instructions and the direct execution of unprivileged instructions, respectively.

6.4.1 Modelling the architecture

An instruction set specifies the transition function of the system's state machine. These transitions are influenced by the platform configuration $c \in C$ (e.g., accessible memory ranges, available hardware extensions, number of cores, interrupt IDs, etc.) and the current machine state $s \in S$ (e.g., registers and memory).

To facilitate verification, we explicitly include the next instruction $i \in I$ in the model—this means the instruction fetch is directly encoded. Interrupts can be modeled as special instructions. The transition function can be formalized as follows:

$$hw : C \times S \times I \to S.$$

Here, the hw function encodes the entire architecture—for instance, in the case of the official

RISC-V Sail model, it spans approximately 16,000 lines of code. When a configuration c is fixed, the hw function can also serve as a simulator. In fact, simulators for Coq and OCaml can be generated directly from the official RISC-V Sail model.

In summary, the hw function provides a high-quality, pre-existing specification of the architecture. In the following, we leverage hw to define the host, virtual, and reference hardware interfaces rather than relying on custom specifications.

6.4.2 Faithful emulation

Privileged instructions (and interrupts) executed by the VM trap to the hypervisor for software emulation. The *virtual hardware interface* constitutes the biggest attack surface that is directly exposed to the software running in the VM. Bugs in the emulation logic can at best disrupt the VM workload, and at worst lead to privilege escalation. We designate the emulation function of a hypervisor as hyp, where $I_p \subset I$ is the set of privileged instructions:

$$hyp : S \times I_p \rightarrow S$$

The hyp function models a single iteration of the *trap-emulate-resume* loop, which is a common pattern in hypervisors. It is important to note that this function can incorporate arbitrary side effects, such as scheduling other virtual machines (VMs).

Based on these definitions, we can formalize the concept of faithful emulation, which ensures that the hypervisor correctly implements the virtual hardware interface:

Definition 6.4.1 (Faithful emulation)**.**

$$\exists c_r \in C, \forall (s,i) \in S \times I_p,\ hyp(s,i) \simeq hw(c_r, s, i)$$

In simple terms, a hypervisor should accurately emulate a reference machine, particularly for privileged instructions.

6.4.3 Faithful Execution

To ensure lock-step execution during *direct execution*, the hypervisor must configure the host hardware to mimic the behavior of the reference machine. The challenge lies in the differences between the configurations of the host and reference machines, as well as the requirement for the hypervisor to manage its own privileged state.

For ease of reasoning, it is useful to separate the machine state into privileged and unprivileged components, i.e., $S = S_p \times S_u$. Moreover, in accordance with virtualization requirements, the privileged state cannot be altered by unprivileged instructions. Therefore, we define a

restricted version of hw that operates exclusively on the unprivileged state:

$$hw|_u : C \times S_p \times S_u \times I \to S_u.$$

Faithful execution is associated with the configuration of the host's privileged state. During the emulation of privileged instructions, the hypervisor may update the VM's privileged state $p_v \in S_p$, typically stored in in-memory data structures, and may also modify the host's privileged state $p_h \in S_p$, which is programmed into the hardware during direct execution.

We define $cfg : S_p \to S_p$ as an abstract hypervisor function that maps a virtual privileged state to the corresponding host privileged state. In practice, this function is often implemented incrementally: when a virtual privileged register changes, the hypervisor updates the associated physical register.

For verification purposes, capturing these incremental updates is sufficient to reconstruct the cfg function. Using this theorem, we formally define faithful execution:

Definition 6.4.2 (Faithful execution)**.**

$$\exists (c_h, c_r) \in C \times C, \forall (p_v, u, i) \in S_p \times S_u \times I, hw|_u(c_h, cfg(p_v), u, i) \simeq hw|_u(c_r, p_v, u, i)$$

In simple terms, the host hardware must be configured to behave as if the VM were running on a machine with a different configuration. Verifying the correctness of this configuration requires instantiating two hardware interfaces (the two "buns" of the hardware burger): one based on the VM platform configuration and its privileged state, and the other based on the host platform configuration and the privileged state derived from the VM's virtual state by the hypervisor.

While the faithful emulation criterion is effective at identifying common implementation bugs, faithful execution goes further by uncovering subtler hardware misconfigurations that are not easily detected through language-level analysis or incomplete hardware specifications.

6.5 Virt-sail, a sail to Rust transpiler

To perform *faithful execution* and *faithful emulation*, we first need to generate a semantically equivalent RISC-V Rust simulator. To the best of our knowledge, there is currently no accurate tool compiling code from Sail to Rust directly. The official repository currently includes a Sail-to-C [80], a Sail-to-OCaml [82] and a Sail-to-Coq [81] transpiler but lacks support for Sail-to-rust.

Previous work [15] introduced Virt-sail [24], a proof of concept Sail-to-rust transpiler that translates Sail code into a semantically equivalent version in rust. This work extends the

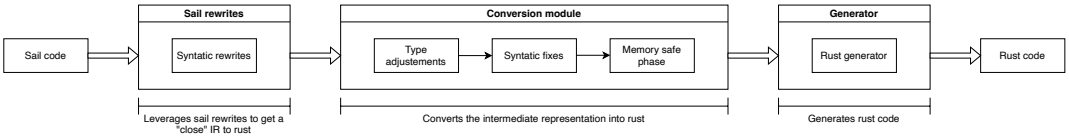

Figure 6.4: Steps inside the Virt-sail compiler

proof-of-concept Virt-sail transpiler that requires much manual patching, even for simple code snippets and successfully generate a subset of the RISC-V sail simulator.

While on the surface Rust can appear similar in design to Sail, underneath they are some semantic gap. For example, Sail contains a primitive named *BitVector* with no direct equivalent in Rust, while Rust's borrow checker, central for ensuring the memory safety garantees in Rust, has no Sail counterpart. These two semantic differences were the key challenge in the transpilation process. To simplify the process, we opted for a multistage compiler design 6.4. Multistage compilers isolate concerns separately, simplifying transformations and optimizations at each step. This reduces overall complexity and makes the system easier to scale and maintain. Virt-sail contains three main stages. It first begins with converting Sail code into "raw invalid" Rust code. Then, it converts the Sail built-in vector type into a custom, equivalent Rust structure (aka BitVector). Finally, it transforms the compiled Rust code into a *borrow checker-friendly* Rust code. Finally, we generate the code.

6.5.1 Sail rewrites

Sail has built-in support for certain syntactic rewrites [28]. These rewrites modify the syntax while preserving the program's semantics. We leverage these transformations and empirically determine the intermediate representation that is *closest* to Rust. In particular, one specific rewrite plays a crucial role. In Sail, function parameters can have multiple real values. At runtime, Sail performs pattern matching across function definitions. One of the available rewrites merges these functions into a single definition. This transformation is significant because it produces an intermediate representation (IR) that is similar to rust Rust, where pattern matching occurs almost exclusively within match statements.

6.5.2 First transformation stage: bitvector stage

The bitvector transformation from Sail to Rust handles the intrinsic differences in how bit vectors and vectors of bits are handled in both languages. In Sail, registers are represented as bit vectors or vectors, which correspond to sequences that lack a direct semantic equivalent in rust. These bit vectors are closely tied to their size and the operations permissible on them. Consequently, converting to Rust requires the generation of a structured semantic representation that accurately reflects both the size and functionality associated with these bit vectors. A direct conversion into usize is prohibited as we loose the semantic equivalence in the overflow behavior. Therefore we create a custom *BitVector* class. In this transformation, we

recursively traverse the entire Rust AST, transforming every node that relates to Sail's built-in bit vectors and replacing them with the custom-generated BitVector struct in our Rust AST. We denote this stage as the BitVector stage.

6.5.3 Second transformation stage: generation of almost valid Rust code

The next stage of the transformation process consists of several smaller modifications to improve both the readability and correctness of the generated code. Currently, the transpiled code contains nested blocks, which, while semantically valid but creates visual clutter and reduce code readability. To address this issue, Virt-sail contains a transformation pass that simplifies these nested structures, resulting in cleaner and more straightforward code. The Sail programming language includes a variety of native functions that we rewrite into their semantically equivalent rust. Rust enumerations need to be properly bound to their corresponding enumeration types, such as *Enumeration::field*. We also encounter issues with overloaded operators that generate illegal function names in rust. To remedy this, we replace the overloaded operators <, =, and > with their respective string representations: *"smaller"*, *"equal"*, and *"bigger"*. We fix the generation of parametric functions by removing unnecessary fields and adding the appropriate import statements and type annotations.

6.5.4 Third transformation stage: borrow checker stage

This pass generates a code that complies with the borrow checker. To do so, it packs the global hardware state (the set of sail registers) in a generated Rust structure of type *SailVirtCtx* and each function in the transpiled sail code contains as first argument an instance of this *SailVirtCtx* labelled *sail_ctx*. Unfortunately, this is not enough to comply with the Rust borrow check as the structure might be moved more than once a a time. The problem here is that the first functions takes an argument of *SailVirtCtx* while another nested call also requires an instance of *sail_ctx*. Although Global variables are technically possible through the *static mut* keyword in Rust and would solve the problem easily, they are highly discouraged as they imply a complete bypass of the Rust borrow checker using the *unsafe* keyword in the entire compiled Rust code. Since memory safety is a guarantee we want to keep, we need another approach. Fortunately we can solve this issue by applying a transformation very similar to the hoisting optimization [74]. We extract all nested functions call and introduce their result with a unique variable uniquely binded to the var_x, where x is incremented by one for each variable.

6.5.5 Generation stage

This stage of the compiler receives the rust intermediate representation and generates the rust code. To do so, it traverses the abstract tree similar to a depth first search. Each type is converted to a string that is used to recursively build the parent type. For example, a *block type* contains a list of *statements*. In the generation phase, each statement will be parsed and

returns a string. The *block* primitives then introduces a bracket phase and generated the list of strings by introducing a semicolon in the middle. The modules also cares about returning a clean indentation. Another option at this stage would be to use the rust formatter directly [**rust-for-Linux**].

6.5.6 Sail modifications

While the official RISC-V SAIL implementation model is sound, it is not complete; for instance, the optional hypervisor extension [37] is not present. However, this extension is required to run the ACE security monitor co-located with Miralis (citation pending for the ACE section). We resolve this divergence by configuring the *HardwareCapability* structure in Miralis appropriately and doing a few manual ajustements in the official specification.

1. **Fix WFI to be Compatible with Two-Lock Execution Step:** The *wfi* (Wait For Interrupt) instruction was modified to work with the two-lock execution step. The sail model changed the mtime register to speedup the execution, which we ultimately don't want for the verification process.
2. **Remove Logic from SFENCE + Add Empty Logic for HFENCE Instructions:** The logic for the *sfence* instruction was removed to streamline the process, while an empty handler for the *hfence* instruction was added. This ensures compatibility with Miralis, where *hfence* does not require additional processing.
3. **Introduce Function *step_interrupt*:** A new function, *step_interrupt*, was added to formalize and verify the virtualization of interrupts.
4. **Remove: Userspace Interrupts in *findPendingInterrupt*:** The code that handled userspace interrupts in the *findPendingInterrupt* function was removed, as this feature is not supported in Miralis.
5. **Introduction: LCOFIE Filter:** The *LCOFIE* filter mechanism was introduced to manage specific interrupt and state transitions in the virtualized environment.
6. **Remove: Tentatively Reserved for User-Level Interrupts Extension:** The code that was tentatively reserved for future support of the User-Level Interrupts extension was removed.
7. **Hardwire Delegation of S-mode operating system Interrupts to 1:** The delegation of S-mode operating system interrupts was hardwired to 1. This ensures that supervisor mode interrupts are automatically delegated.

Additionally, we enforce a series of preconditions necessary for Miralis to begin execution. These include verifying the presence of userspace mode and enforcing a 64-bit machine environment. The generated code contains around 6,000 lines of rust code.

Algorithm 5 Create New Symbolic Contexts

```
1: miralis_context ← GenerateSymbolicMiralisContext
2: sail_context ← convert_miralis_to_sail(miralis_context)
3: return (miralis_context, sail_context)
```

6.6 Symbolic verification in Miralis

We verify Miralis using symbolic execution. Hardware instructions don't have unbounded loops or constructs that cause trouble for the symbolic verification. By definition hardware state is finite and each instruction finishes in a finite amount of time (and quickly). Therefore symbolic verification is an ideal technique. Unlike traditional testing, which is limited to specific inputs, formal verification provides strong mathematical guarantees of equivalence, ensuring the VFM behaves *equivalently* to the official RISC-V specification [83], by exhaustively exploring all possible states. It therefore offers a complete coverage and naturally includes adversarial edge cases and uncommon configurations.

Algorithm 6 Lightweight formal method procedure in Miralis: Mret example

```
1: (miralis_context, sail_context) ← symbolic_contexts() ▷ Generates the symbolic contexts
2: emulate_MRET(miralis_context)
3: execute_MRET(sail_context)
4: assert miralis_context ≡ sail_to_miralis(sail_context)
```

We introduce a new folder in the codebase named model_checking, dedicated to formal verification tasks. We add the the generated Rust RISC-V simulator and the external hand-written Sail prelude with a series of tests to ensure the correctness of the implementation. It starts by generating two contexts, we denote as $context_{miralis}$ and $context_{reference}$, which corresponds to the internal state of the Miralis hypervisor and the RISC-V reference. On the next line, we transform both the $context_{miralis}$ and $context_{reference}$ by applying the corresponding function, leading to $context_{miralis} = f^{miralis}(context_{miralis})$ and $context_{sail} = f^{sail}(context_{sail})$. Finally, we convert the sail state back to miralis state and check whether or not the context are equivalent. If the equivalence relationship fails, then there is a divergence between the two implementation. According to lock step execution 6.3, the transitions are *atomic* and there is therefore no need to reason about intermediate states during the transition, which leads to the following equivalence relaionship 6.6.1.

We use Kani as the symbolic execution framework and we found 25 bugs during the execution of Kani both in handling the the privileged instructions as well as in the interrupt virtualisation process. The interrupts were not delivered in the correct order according to the RISC-V specification. Figure 6.1 shows the execution time for a macbook containing the m3 chip.

Definition 6.6.1 (Equivalence verification)**.**

$$f^{miralis}(ctx_{miralis}) \simeq sail\_to\_miralis(f^{sail}(sail\_to\_miralis(ctx_{miralis}))$$

Table 6.1: Verification time of the emulation pipeline

Verification Task		Time	
`mret` instruction	68s	`wfi` instruction	28s
`sret` instruction	56s	Instruction decoder	45s
`CSR` read	99s	Interrupt virtualization	94s
`CSR` write	9min	End-to-end emulation	118min

6.6.1 Symbolic verification of illegal instructions

We start by verifying the emulation of illegal instructions (MRET, SRET, WFI, SFENCE and the CSR operations). To make the process a bit easier, we leverage the fact that Miralis and the SAIL both contain two functions, *readCSR* and *writeCSR*, which read from and write to the corresponding CSRs while maintaining the invariants (for example, if a register is read-only, any write to it will be ignored). This allows us to verify the correctness of the register decoder component at the same time. Next, we formally verify the behavior of the MRET and WFI instructions.

Through the formal verification process, we identified a series of issues and implemented adjustments to align with the formal RISC-V specification. Additionally, we added preliminary support for the optional RISC-V vector and cryptographic extensions [50, 49]. The following bugs and updates were addressed:

1. **mret: Set MPP to U-mode after mret**
   When executing the *mret* instruction, the *MPP* (Machine Previous Privilege) field should be set to User mode (U-mode) if the system is returning to user space.
2. **mret: Default MPP value should be M-mode if U-mode is not enabled**
   If User mode is not enabled, the *MPP* field should default to M-mode after *mret*. This ensures that when returning from a lower privilege mode, the system correctly maintains the expected privilege level.
3. **mvendorid: Register must have only 32 bits**
   The *mvendorid* CSR, which identifies the vendor of the processor, must be a 32-bit register.
4. **scounteren, mcountinhibit, mcounteren: Registers must have 32 bits**
   These counter-related CSRs (*scounteren*, *mcountinhibit*, *mcounteren*) are specified to be 32-bit registers.
5. **mepc, sepc: The least significant bits must be hardwired at 0**
   The *mepc* (Machine Exception Program Counter) and *sepc* (Supervisor Exception Program Counter) registers must have their least significant bits (LSB) cleared (set to 0). This ensures proper alignment of the program counter.
6. **mtvec, stvec: Restrict to valid vector modes (direct or vectored)**
   The *mtvec* (Machine Trap Vector) and *stvec* (Supervisor Trap Vector) registers must only

support valid vector modes: either direct or vectored.

7. **medeleg: Bit 11 is read-only zero**
   The *medeleg* (Machine Exception Delegation) register, specifically bit 11, must always read as zero.
8. **satp: Discard writes with invalid mode**
   The *satp* (Supervisor Address Translation and Protection) register should discard any writes with an invalid mode. This ensures that the system respects valid address translation modes and prevents invalid configurations.
9. **CSR read: Return 0 instead of panicking on unknown CSR**
   When attempting to read from an unknown or unsupported CSR, the hypervisor should return a trap rather than causing the system to panic.
10. **vstart: Invalid write mask**
    The *vstart* register, which controls the starting index of vectorized operations, should not accept invalid write masks.
11. **mcause, scause: Allow any value**
    The *mcause* (Machine Cause) and *scause* (Supervisor Cause) registers should be allowed to hold any value, reflecting the full range of possible exception causes.
12. **pmpcfg: Odd PMP (Physical Memory Protection) Config registers should be invalid**
    In the *pmpcfg* (Physical Memory Protection Configuration) register set, odd-numbered configuration registers are reserved and should be marked as invalid. Any access to these registers should result in an exception or error.
13. **pmpaddr: Invalid legalization mask**
    The *pmpaddr* registers, used for specifying physical memory protection addresses, must have a valid "legalization mask". Any writes that do not conform to this mask should be considered invalid and ignored.
14. **pmpaddr: W = 1 & R = 0 is reserved**
    In the *pmpaddr* registers, the configuration where Write (*W*) is enabled and Read (*R*) is disabled is reserved and should not be allowed. Writes to these registers should be rejected if they attempt to set this illegal combination.
15. **mepc, sepc: Enforce alignment based on C extension**
    The *mepc* and *sepc* registers, as part of the exception handling mechanism, must be aligned properly according to the C extension rules.
16. **mie: Writes were discarding**
    The *mie* (Machine Interrupt Enable) register, responsible for enabling machine-level interrupts, was discarding writes in certain cases.
17. **sie, sip: Filter based on mideleg**
    The *sie* (Supervisor Interrupt Enable) and *sip* (Supervisor Interrupt Pending) registers must be filtered based on the *mideleg* (Machine Interrupt Delegation) setting. This ensures that only interrupts allowed by the machine mode delegation are handled by the supervisor mode.
18. **Decoder**
    The two decoders were not deconding the same set of instructions. Our implementation

was detecting some instructions as *MRET* or SFENCE while they were in reality unknown instructions in the sail decoder.

19. **Bypass CSR R/W with X0**
    If we want to write to a read only register, we should not read and write the CSR register on top of it. Doing a read / write on some CSR modifies the underlying value afterwards.
20. **SailBugfix: Divide pmp_csr_idx / 2 in pmpcfg write**
    We don't want to write to more than the available number of virtual registers. However, the *pmpcfg* registers have a stride of 2. Therefore, we need to divide first the index by two, and then we can check for overflow.
21. **Wrong immediate offset in compressed load and stores**
    In compressed load and compressed store instructions, the bit at position two comes from the raw bit at position 6. Therefore, we had to shift it by 4 instead of 3. On top of that, we were shifting by two ot three bits, which as not requested in the specification.
22. **Unsigned loads with 8 bytes are not decoded**
    Unsigned loads with 8-byte precision were not decoded and corresponded to an unknown instruction in Miralis.

6.6.2 Symbolic verification of the interrupts

In the trap handling logic, Miralis first checks if an interrupt is available, and if so, it injects the interrupt into the virtual firmware mode. We refer to this process as interrupt virtualization, as the interrupts are virtually injected by Miralis and are not native to the processor. We verify this logic against the corresponding sail function and make sure we inject the interrupts correctly to the firmware. During symbolic execution, we discovered a bug where the interrupts were not injected in the correct order. The snippet shows the fix, which ensures the interrupts are injected according to the official RISC-V order.

```
fn find_pending_interrupt_by_priority(ip: usize) -> Option<usize> {
    if ip & mie::MEIE_FILTER != 0 {
        Some(MEIE_OFFSET)
    } else if ip & mie::MSIE_FILTER != 0 {
        Some(MSIE_OFFSET)
    } else if ip & mie::MTIE_FILTER != 0 {
        Some(MTIE_OFFSET)
    } else if ip & mie::SEIE_FILTER != 0 {
        Some(SEIE_OFFSET)
    } else if ip & mie::SSIE_FILTER != 0 {
        Some(SSIE_OFFSET)
    } else if ip & mie::STIE_FILTER != 0 {
        Some(STIE_OFFSET)
    } else {
        None
    }
}
```

Listing 6.1: Detection of the interrupts in the correct order

6.6.3 Symbolic verification of traps

Verifying trap is slightly harder because Miralis uses given by the hardware and store this in a *trap_cause* data structure. Miralis uses the values inside the *trap_cause* data structure to prepare the trap injection process. Therefore we need to initialise the *trap_cause* structure before we run the symbolic verification. Fortunately, we can use the same function of the sail specification to first fill the *trap_cause* data structure and then perform the formal verification. During the verification of the trap, we found no bugs.

6.6.4 Symbolic verification of the PMP installation in hardware

One of the key point to formally verify it to assert is that the exposed virtual hardware interface is a variant the host hardware interface as shown in the hardware burger figure 6.1. Miralis contains virtual PMPs that are installed and removed when transitioning from the firmware to the payload and vice-versa. The official RISC-V implementation contains the function pmpCheck that detects illegals read, writes and execute when given a set of pmp register values. We formally verify that if the firmware protects a memory location, the change will be successfully propagated physically.

7 Evaluation

We evaluate Miralis to answer the following key questions:

- Is Miralis backward compatible with the existing software stack (OpenSBI, U-boot, the Linux kernel,....)?
- What is the performance of Miralis on cpu, disk and network microbenchmarks?
- What is the performance of Miralis on application workloads?

7.0.1 Experiment setup

We run Miralis on the VisionFive2 board and Premier P550 board. The VisionFive2 board is a high-performance RISC-V single-board computer featuring the StarFive JH7110 SoC, which includes a quad-core RISC-V U74 CPU with 2MB L2 cache, operating up to 1.5GHz. It offers 4GB of LPDDR4 RAM. The Premier P550 board board is a single-board computer equipped with an ESWIN EIC7700X SoC, containunt 2MB L2 cache and 1.8GHz. It comes with 16GB of LPDDR4 RAM. We run the VisionFive2 board using OpenSBI v1.0, and the Linux kernel using version 5.15 and the Premier P550 board using OpenSBI v1.4 and Ubuntu 20.4. We count the number of trap and world switches by introducing a driver communicating with Miralis in the kernel. Benchmarks involving a remote client runs on a 8 cores CPU running at a frequecy of 3.5 GHZ and running ubuntu and the two machines are connecting using the local ethernet.

7.0.2 Seamless Firmware Virtualization

Miralis runs with unmodified vendor firmware images, which makes Miralis backward compatible with the current boot stack. By current boot stack, we assume the standard boot procedure is to first run U-Boot SPL, followed by OpenSBI before jumping to S-mode and finally launch U-boot followed by the Linux kernel as shown in figure 7.1. We insert Miralis between U-boot SPL and OpenSBI in the boot flow. We install Miralis after OpenSBI and U-boot in the address space on both the VisionFive2 board and the Premier P550 board. We configure the first stage bootloader to jump on Miralis instead of OpenSBI. Finally Miralis

	VisionFive2 board	Premier P550 board
Number of cores	4	4
Frequency	1.5GHz	1.8GHz
L1 cache	32KB	32KB
L2 cache	2MB	256KB
L3 cache	None	4M
RAM	4GB	16GB
OpenSBI version	1.0	1.4
Linux kernel version	5.15	6.6

Table 7.1: Characteristics of the VisionFive2 board and the Premier P550 board

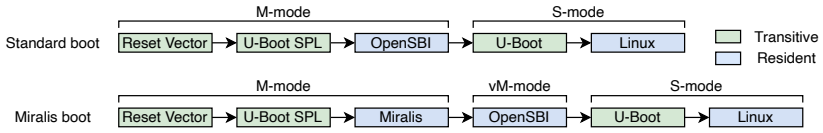

Figure 7.1: Boot process

jumps to OpenSBI and handles the transitions between the firmware and the payload.

7.0.3 Cost of firmware exceptions

We begin by measuring the overhead introduced by exceptions in Miralis, which has two categories of transitions. The first category corresponds to the cost of handling firmware exceptions within the firmware itself. The second accounts for the cost of a round-trip between the payload and the firmware when the payload requests a firmware-specific operation. Table 7.2 presents the cost of a firmware exception and a world switch on the Spike simulator, the VisionFive2 board and the Premier P550 board. Figures 7.2 and 7.3 classifies exceptions occurring between the payload and the firmware, showing that most exceptions from the payload to the firmware do not correspond to *firmware-specific operations* but rather to *hardware emulation logic*, which emulates features ratified but not yet available in hardware, such as the *sstc extension* or the *time* register. These operations do not reside in the firmware because they inherently require firmware handling but because they must execute in the highest privileged mode, meaning they could theoretically reside in the security monitor instead. Therefore we can implement those operations directly in Miralis and this introduce a third category of transitions, namely payload $\rightarrow$ Miralis $\rightarrow$ payload, which we assume does not introduce additional overhead.

7.0.4 Microbenchmarks

We evaluate Miralis on a set of three microbenchmarks. We use CoreMarkPro as a CPU intensive workload, Iozone as a disk intensive workload and finally Memcached to evaluate the

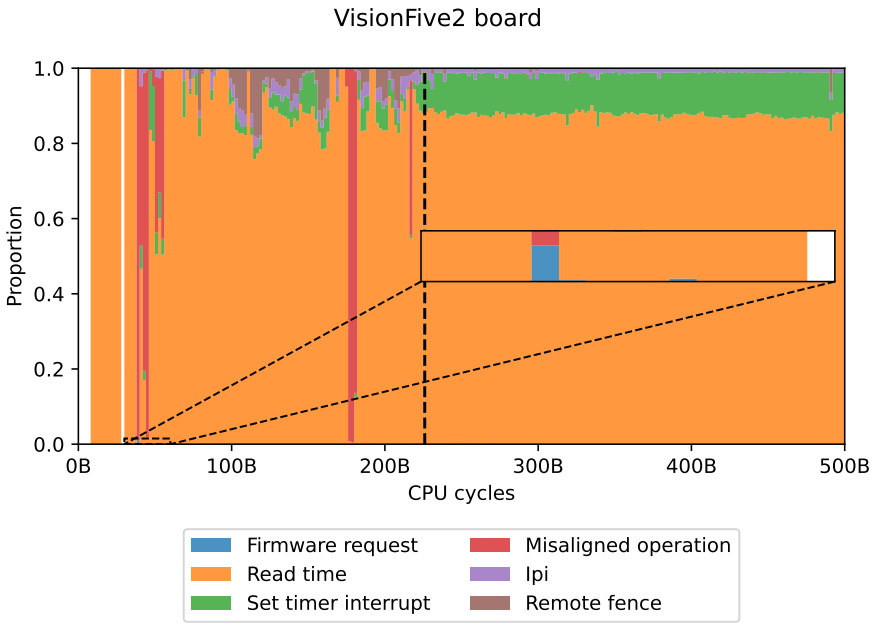

Figure 7.2: Distribution of firmware exceptions on the VisionFive2 board

	Firmware trap	World switch
VisionFive2 board	483	2704
Premier P550 board	271	4098
Spike	396	2606

Table 7.2: Cost of firmware exceptions in cycles

impact on network latency. We evaluate two versions of Miralis where the first one corresponds to Miralis and the operating system is isolated from the firmware. On the second one, we also run the same configuration but with Miralis implementing the *hardware emulation logic* on top of the strict protect payload policy.

CPU microbenchmark We start by evaluating the performance implications of Miralis on a CPU-intensive workload. We evaluate the impact of running CoreMarkPro, on all available cores. Figures 7.4 and 7.5 shows that across the 8 microbenchmarks, the number of iterations per second degrades by 1% to 4% on the VisionFive2 board and 2.4% to 10.3% on the Premier P550 board, while introducing no overhead on both platforms when Miralis directly emulates the missing hardware features. The number of exceptions per second is around 16.7k on the VisionFive2 board and 15.9k on the Premier P550 board which corresponds to a workload with few exceptions and explains the generally low overhead. The difference between the two platforms comes mainly from the fact that the Premier P550 board receives the page faults in the firmware.

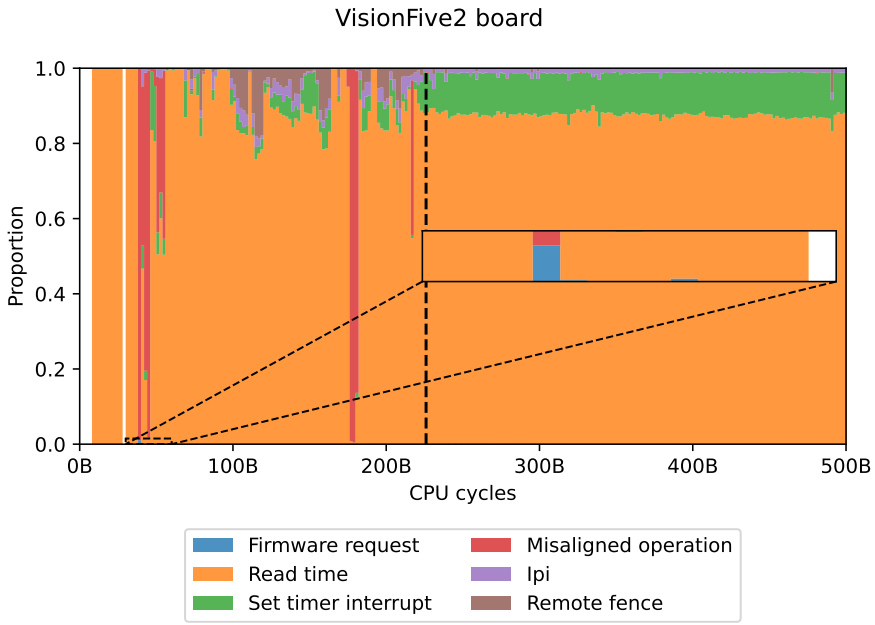

Figure 7.3: Distribution of firmware exceptions on the Premier P550 board

Disk microbenchmark Next, we investigate I/O performance by running Iozone on top of Miralis for both reads and writes. We observe a larger number of firmware exceptions on both the VisionFive2 board and the Premier P550 board. We use a record length of 128KB and vary the disk size, spanning from 128KB to 128MB. Figures 7.6 and 7.7 shows that the throughput in KB/s degrades by 7.8% to 13.5% on the VisionFive2 board and by 27% to 56.3% on the Premier P550 board for reads and 2.1% to 8.4% on the VisionFive2 board and by to on the Premier P550 board for writes. Compared to the CPU microbenchmark, the difference in throughput degradation comes from the higher number of exceptions to the firmware, which is 21K on the VisionFive2 board and 62K on the Premier P550 board. Similarly, when Miralis implements the hardware emulation, we observe no degradation in throughput due to the negligible firmware exceptions on the VisionFive2 board. On the Premier P550 board, we see that the latency suffers a bit at the beginning but is then similar.

Network latency Finally, we run a network latency microbenchmark to investigate, whether or not Miralis introduces an extra latency with respect to the network. To measure network performance, we run Memcached on top of Miralis and measure the latencies of both reads and writes using Memtier [69] from a remote machine. Figures 7.8 and 7.9 shows that the read latency increases from 7.4ms to 14.9ms in the 50th percentile and from 228ms to 288ms in the 99th percentile, leading to 101% and 26.3% overhead in latency on the VisionFive2 board. Similar to the disk microbenchmark, we observe a high number of exceptions, justifying the performance degradation. Implementing hardware emulation in Miralis returns latencies of 6.7ms and 228ms, respectively. The 0.7ms difference comes from the fact that Miralis

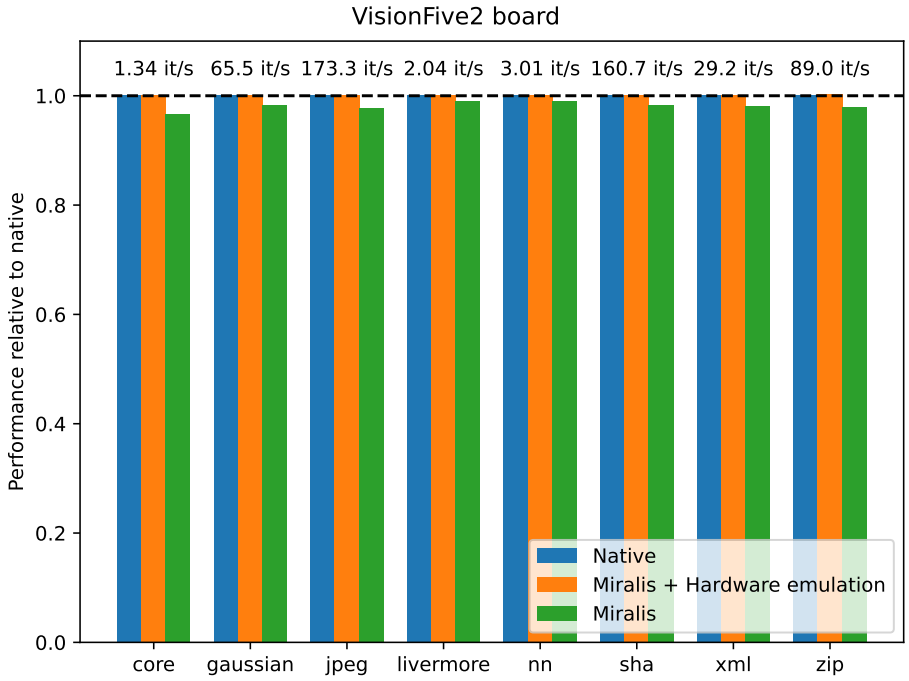

Figure 7.4: Coremarkpro on the VisionFive2 board

has an optimized path for the most frequent firmware exception, reading the CLINT for the unsupported *time* register. The write curve follows the same distribution. On the Premier P550 board, we observe a similar tendency with an increase from 7.4ms to 14.9ms in the 50th percentile and from 228ms to 288ms in the 99th percentile, leading to 101% and 26.3% overhead while the hardware emulation gives a latency of 7ms and 226ms.

7.0.5 Application benchmarks

In this section, we run a series of applications benchmarks, which allows us to measure the performance of Miralis on a set of real applications. We measure the throughput on three databases (Redis, Memcached and MySQL) and the compilation duration of two well-known softwares, Redis and the Linux kernel. Similar to the microbenchmark section, we evaluate Miralis with the strict protect payload policy with and without support for hardware emulation.

Memcached and Redis: We use *Redis* and *Memcached* as representative of two network-intensive server applications. We use the YCSB benchmark suite, running *workloadA* and 1000000 operations on *Redis v7.0.15* and also 1000000 operations *Memcached v1.6.24*; Figures 7.10 and 7.11 reports the results. For Redis, we observe a degradation of 44% for Memcached and 62% on the VisionFive2 board and 72.2% and 53.3% on the Premier P550 board. We explain this clear degradation by a very large amount of exceptions to the firmware. Miralis recorded 276K and 390K firmware exceptions on the VisionFive2 board and 264K, 381K on the Premier P550 board. Implementing the logic directly in Miralis improved the throughput by

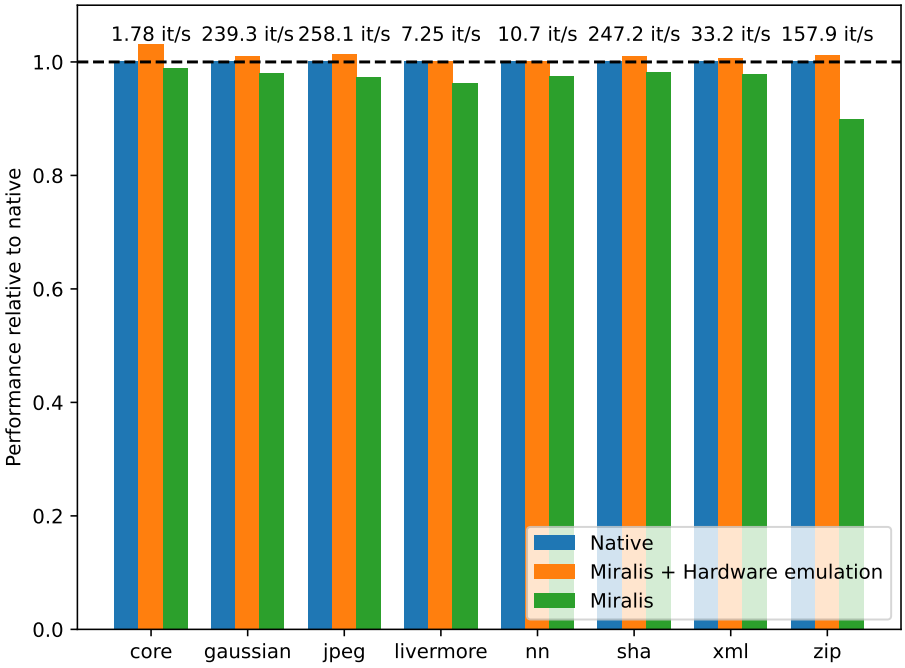

Figure 7.5: Coremarkpro on the Premier P550 board

6.2% and 7.6% respectively for Redis and Memcached on the VisionFive2 board and -0.04% and 1.2% on the Premier P550 board. The reason for that is both workloads causes a lot of exceptions because the *time* register isn't delegable to userspace and Miralis emulates the read of time faster than OpenSBI.

MySQL: Next, we use MySQL version v8.0.41 as a representative of network and disk database. Redis and Memcached interact less with the disk than a traditional database as MySQL is a relational database where values are stored in the disk instead of Memory. We benchmark MySQL using workload *oltp_read_write* from Sysbench using 128 threads a duration of 5 minutes. Figures 7.10 and 7.11 shows a throughput degradation of 16.5% on the VisionFive2 board and 43.1% on the Premier P550 board compared to the baseline and an improvement of 4.3% on the VisionFive2 board and 2.5% on the Premier P550 board when Miralis emulates the hardware directly. Since MySQL is a mixture of network operations and disk operations, we observe overall less firmware exceptions which explains the lower spread in the results with a total of 162K on the VisionFive2 board and 302K on the Premier P550 board.

Compiling Redis and the Linux kernel: We compile the Redis database and the Linux kernel, which represent two CPU and Memory intensive workloads. We compile *Redis v7.4.2* and *Linux v6.14* using all available cores, namely 4 for the Visionfive2 board and the premier P550 board. Figures 7.10 and 7.11 shows a degradataion of 2.8% on the VisionFive2 board and 12.5% on the Premier P550 board in latency without hardware emulation in Miralis and exactly the same speed when Miralis emulates the hardware. Since the benchmarks are CPU intensive, they cause a similar number of exceptions than the cpu microbenchmark and thus lead to a

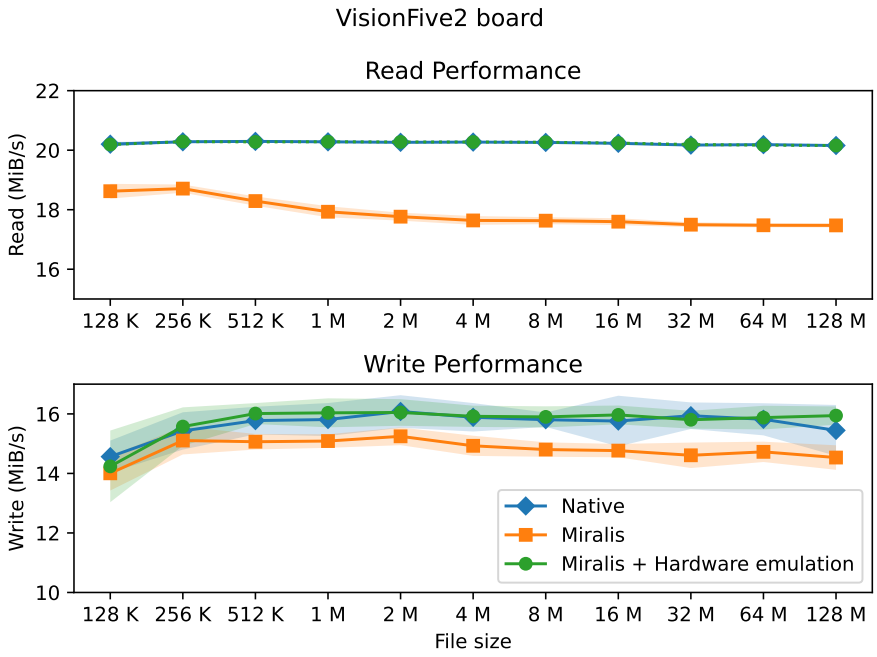

Figure 7.6: Iozone on the VisionFive2 board

low variation in the results.

7.0.6 Keystone

To conclude the experiment section, we measure the performance of the Keystone reimplementation inside Miralis. Similar to Keystone[101], we measure the Keystone enclave using the RV8 microbenchmarks. We compare the performance of the enclave against the same workload in Miralis only (without the enclave) and do not implement hardware emulation in Miralis. Figure 7.12 shows a relative performance range from 3% to 0% on the firmware with a mean value of 1%, when running inside the enclave. This corresponds to the overhead introduced when switching between the enclave and the rest of the world using the *pause and resume* logic in Keystone.

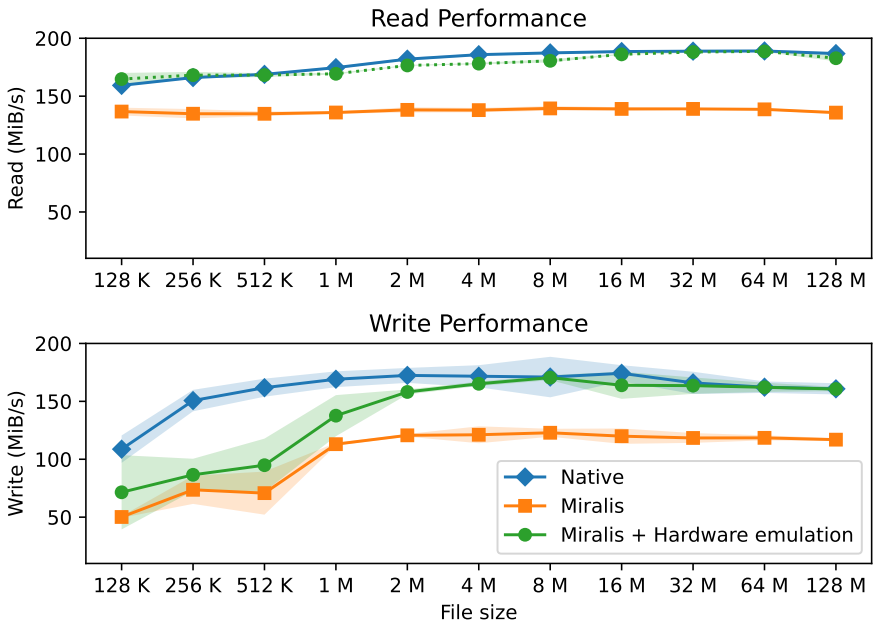

Figure 7.7: Iozone on the Premier P550 board

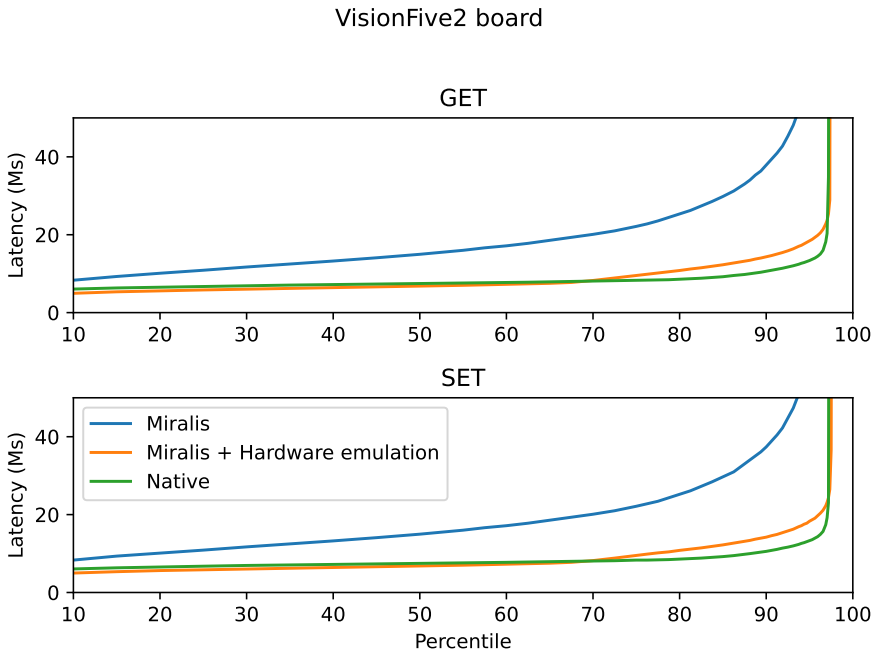

Figure 7.8: Network microbenchmark on the VisionFive2 board

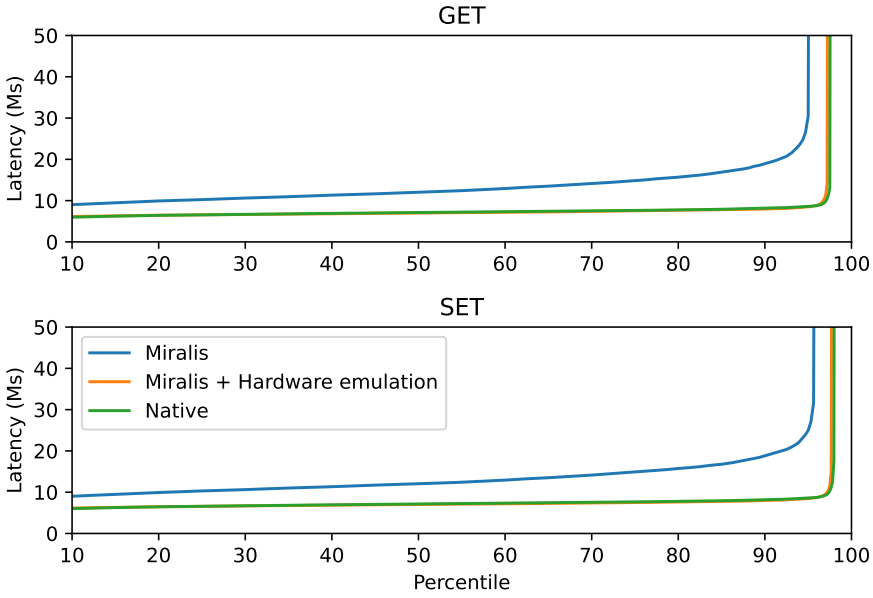

Figure 7.9: Network microbenchmark on the Premier P550 board

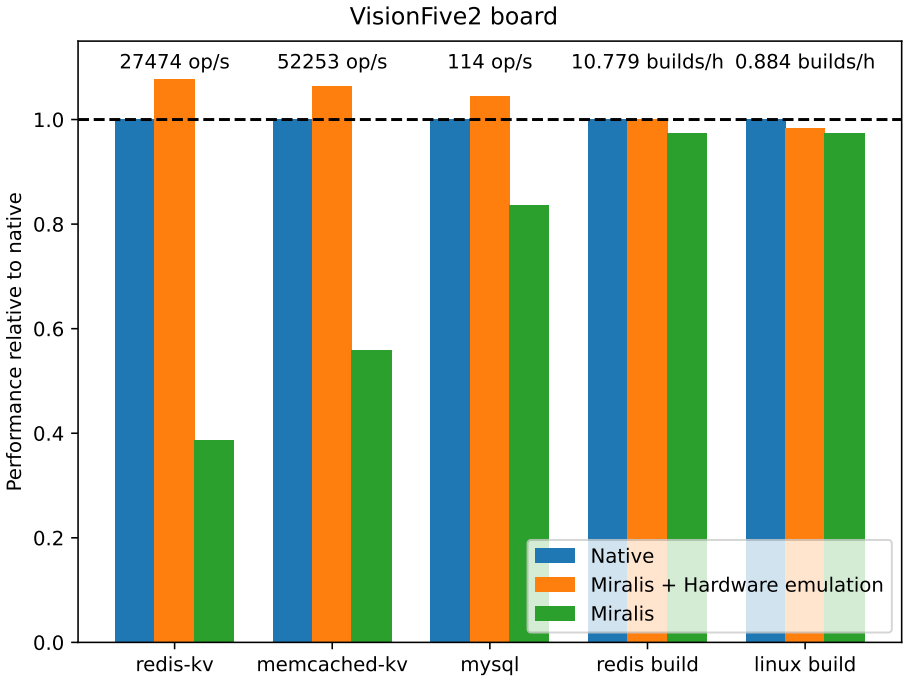

Figure 7.10: Application benchmarks on the VisionFive2 board

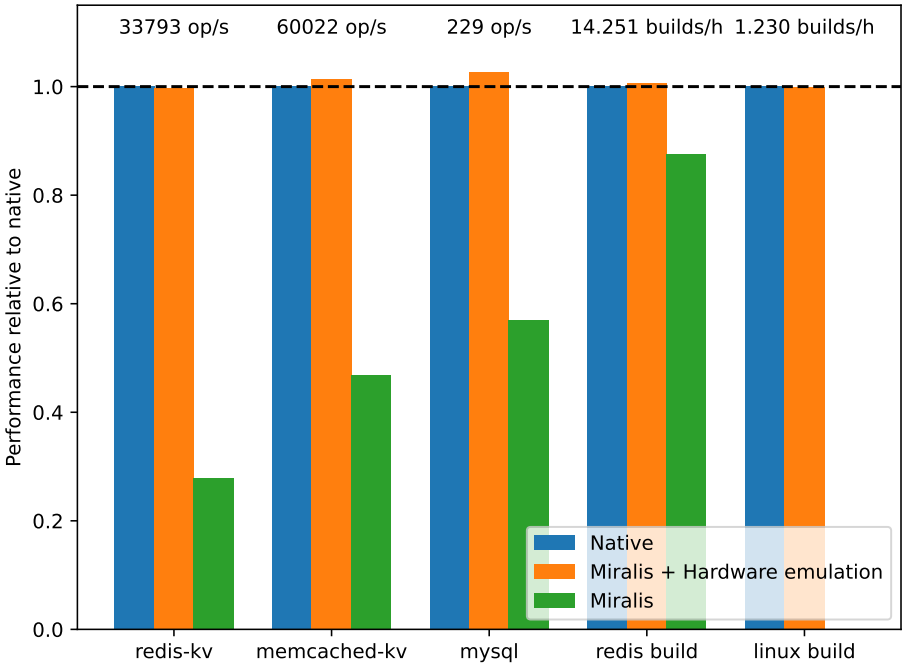

Figure 7.11: Application benchmarks on the Premier P550 board

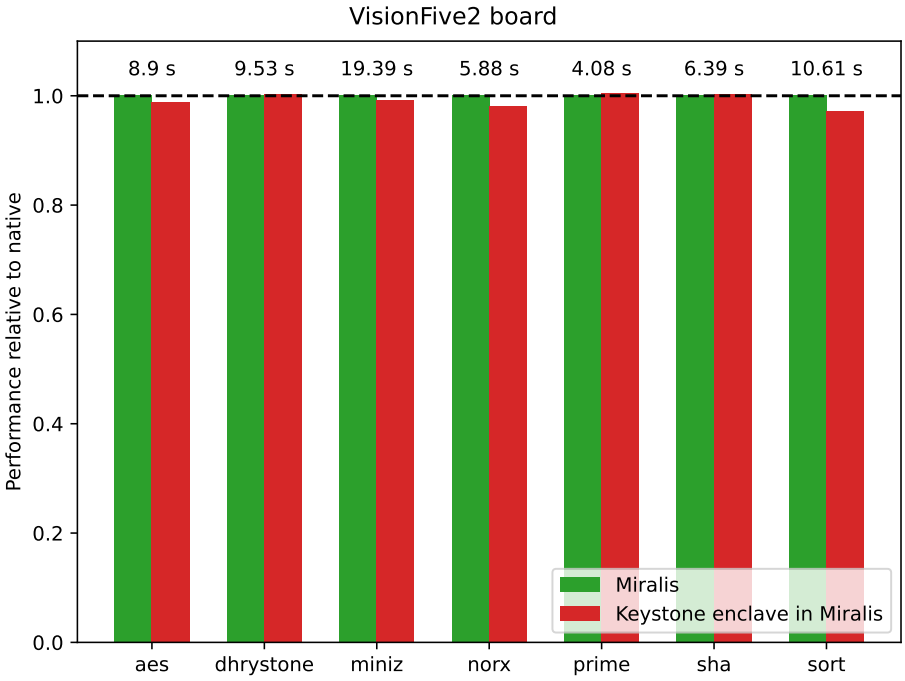

Figure 7.12: RV8 microbenchmark on Keystone

8 Conclusion and future work

8.1 Next steps in Miralis

There are a few follow-up tasks that could strengthen Miralis for a potential SOSP or OSDI publication. In the area of lightweight formal methods, there is an opportunity to further improve coverage, particularly in handling misaligned loads and stores as well as the virtual CLINT driver. The measurement section for the Keystone policy could be improved by incorporating an I/O-intensive application such as Iozone.

8.2 Future ideas

8.2.1 DAP Server:

A Debug Access Port (DAP) server is a software component that allows external debugging tools to communicate with a target system's hardware, such as a microprocessor or an embedded system. It serves as an interface between debugging tools and the system, enabling tasks like memory inspection, register access, execution control, and remote debugging of embedded systems.

In the context of Miralis, the firmware hypervisor can act as a remote debugging tool for real hardware such as the VisionFive2 board or Premier P550 board. A DAP server within Miralis would allow developers to remotely access, analyze, and debug firmware running on physical hardware using popular IDEs like VS Code. This would provide a robust debugging experience directly on real hardware, complementing emulators such as Spike or QEMU, which are inherently simpler and unable to expose a wide range of hardware-specific bugs.

8.2.2 Miralis as a Library:

Another potential future development for Miralis is its transformation into a modular library, enabling integration with security monitors beyond ACE and Keystone. Currently, Miralis

runs with ACE or Keystone as a single binary blob. Developing an API would allow Miralis to function as a standalone library that other or future security monitors can leverage. For example, ACE delegates tasks such as inter-processor interrupts (IPIs), interrupt handling, and operations like misaligned load/store emulation to Opensbi. By converting Miralis into a library, it could replace Opensbi in these scenarios, offering a flexible and reusable solution for other security monitors. In the future, security developers could design their own security monitors while delegating exception handling to Miralis.

8.2.3 Integration with Other Security Monitors

Tyche: Tyche [14] is a security monitor that provides a framework for isolating and managing Trust Domains, which are secure execution environments running on commodity hardware. It enforces strong isolation, confidentiality, and resource control for multiple TDs, offering a flexible, capability-based API for their management. Core isolation in Tyche ensures that each TD is assigned to specific cores, preventing interference between TDs on the same physical CPU. Tyche enables secure multi-tenancy and confidential computing in environments where hardware-level isolation mechanisms, such as Intel SGX or AMD SEV, may be unavailable or overly restrictive. Since Tyche requires firmware as part of the trusted computing base (TCB), incorporating Miralis into its TCB could significantly enhance its security properties.

Penglai Enclave: The Penglai Enclave [32] is an open-source trusted execution environment (TEE) designed for the RISC-V architecture, presented at OSDI '21. It provides a secure, scalable, and high-performance solution for security-sensitive applications, ensuring the confidentiality and integrity of data and code execution. It has been widely adopted by leading Chinese institutions such as Huawei and StarFive. The Penglai Enclave is a strong candidate for integration with Miralis, since the firmware is part of the root of trust.

8.2.4 Extensions to Other Architectures, Devices, and Chips

Other Architectures: Miralis currently virtualizes firmware exclusively on the RISC-V architecture. RISC-V is an ideal choice for research, being relatively simple (900 pages of reference documentation compared to ARM's 15,000) and strictly virtualizable with respect to the Popek and Goldberg criteria. However, the two most widely used ISAs today are x86 and ARM. Porting Miralis to one of these architectures would be a natural step toward broader adoption.

Other Devices: At present, Miralis protects only the CLINT. The firmware however, retains access to many other devices commonly found in real systems, such as UART, NIC, and GPU. Although the kernel is isolated from the firmware the firmware could still exploit these devices to leak information or manipulate kernel control flow. Extending Miralis to secure additional devices represents an important avenue for future work.

Other Chips: Fiedler et al. [34, 35] introduced the concept of *the de facto OS*, arguing that an operating system is just one component of a broader system. Modern boards integrate

numerous peripherals, many of which rely on closed-source firmware. As a CPU-centric firmware monitor, Miralis cannot mitigate vulnerabilities in these peripheral devices. However, many auxiliary chips could benefit from Miralis. Exploring the security implications of these chips on modern boards remains an open research direction.

8.3 Final Words

This thesis first introduced the concept of security policies. The first policy, *Strict Protect Payload Policy*, aims to completely isolate the payload from the firmware. This thesis demonstrated that it successfully prevents the firmware from interfering with the operating system. The guarantees provided by this policy were meticulously documented. The second policy, *ACE Policy*, facilitates the creation of confidential virtual machines that exclude both the hypervisor and the firmware from the trusted computing base. This was achieved by colocating Miralis with the ACE security monitor and introducing the concept of *security monitor context switching*. The implementation involved running the VM-mode firmware on top of Miralis and the rest on top of ACE. During transitions between these components, the system transfers the entire security monitor context, granting full system control to the active monitor.

Beyond these policies, this thesis brings the Virt-Sail compiler to a functional state where it reliably compiles a subset of Sail code. This enabled the generation of a RISC-V simulator and the verification of the emulation logic using *lightweight formal methods* inside of Miralis, revealing 25 bugs in the Miralis codebase.

Finally, this thesis measured the overhead introduced by running the firmware in a deprivileged userspace context on two different platforms. Through a series of rigorous measures, this thesis demonstrates that running the firmware in userspace introduces no overhead at runtime.

Bibliography

[1] M. Accetta et al. “Mach: A New Kernel Foundation for UNIX Development”. In: *Proceedings of the 1986 Summer USENIX Conference*. 1986, pp. 93–112.

[2] Sten Agerholm and Peter Gorm Larsen. “A Lightweight Approach to Formal Methods.” In: 1998, pp. 168–183.

[3] Paul Alcorn. *AMD Issues Fix and Workaround for Ryzen's fTPM Stuttering Issues*. https://www.tomshardware.com/news/amd-issues-fix-and-workaround-for-ftpm-stuttering-issues. 2022.

[4] Balena. *balenaEtcher - Flash OS images to SD cards USB drives*. Accessed: 2025-02-07. 2025. URL: https://www.balena.io/etcher/.

[5] P. Barham et al. “Xen and the Art of Virtualization”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. 2003, pp. 164–177.

[6] Andrew Baumann. “Hardware is the new Software.” In: 2017, pp. 132–137.

[7] Dana Behling. *Bring Your Own Backdoor: How Vulnerable Drivers Let Hackers In*. https://blogs.vmware.com/security/2023/04/bring-your-own-backdoor-how-vulnerable-drivers-let-hackers-in.html. 2023.

[8] Brian N. Bershad et al. “Extensibility, Safety and Performance in the SPIN Operating System”. In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*. Dec. 1995.

[9] *Boot Unguarded: x86 Trust Anchor Downfalls to The Leaked OEM Internal Tools and Signing Keys*. https://hardenedlinux.org/blog/2023-09-07-boot-unguarded-x86-trust-anchor-downfalls-to-the-leaked-oem-internal-tools-and-signing-keys/. 2023.

[10] James Bornholt et al. “Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3.” In: 2021, pp. 836–850.

[11] Davide Bove and Lukas Panzer. *R5Detect: Detecting Control-Flow Attacks from Standard RISC-V Enclaves*. 2024. arXiv: 2404.03771 [cs.CR]. URL: https://arxiv.org/abs/2404.03771.

[12] Per Brinch Hansen. “The Nucleus of a Multiprogramming Operating System”. In: *Communications of the ACM* 13 (1970), pp. 238–250. DOI: 10.1145/362258.362271.

[13] Edouard Bugnion et al. “Bringing Virtualization to the x86 Architecture with the Original VMware Workstation”. In: *ACM Trans. Comput. Syst.* 30.4 (Nov. 2012). ISSN: 0734-2071. DOI: 10.1145/2382553.2382554. URL: https://doi.org/10.1145/2382553.2382554.

[14] Charly Castes et al. “Creating Trust by Abolishing Hierarchies”. In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HOTOS ’23. Providence, RI, USA: Association for Computing Machinery, 2023, pp. 231–238. ISBN: 9798400701955. DOI: 10.1145/3593856.3595900. URL: https://doi.org/10.1145/3593856.3595900.

[15] Charly Castes et al. “Kicking the Firmware Out of the TCB with the Miralis Virtual Firmware Monitor”. In: *Proceedings of the 2nd Workshop on Kernel Isolation, Safety and Verification*. KISV ’24. Austin, TX, USA: Association for Computing Machinery, 2024, pp. 8–15. ISBN: 9798400713019. DOI: 10.1145/3698576.3698764. URL: https://doi.org/10.1145/3698576.3698764.

[16] Haogang Chen et al. “Using Crash Hoare Logic for Certifying the FSCQ File System”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 18–37.

[17] Haogang Chen et al. “Verifying a High-Performance Crash-Safe File System Using a Tree Specification”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 270–286.

[18] Pau-Chen Cheng et al. “Intel TDX Demystified: A Top-Down Approach.” In: *ACM Comput. Surv.* 56.9 (2024), 238:1–238:33.

[19] openSUSE Community. *openSUSE Tumbleweed RISC-V JeOS Image for VisionFive2*. https://download.opensuse.org/ports/riscv/tumbleweed/images/openSUSE-Tumbleweed-RISC-V-JeOS-starfivevisionfive2.riscv64.raw.xz. Repository: https://download.opensuse.org/. 2024.

[20] RISC-V Community. *RISC-V AP-TEE*. https://github.com/riscv-non-isa/riscv-ap-tee. Accessed: 2024-12-05. 2024.

[21] RVspace Community. *VisionFive2 Debian RISC-V Image (2023-02 Release)*. https://rvspace.org/en/project/VisionFive2_Debian_Wiki_202302_Release. Resource available from RVspace. 2023.

[22] The Linux Kernel Community. *Linux Kernel Virtual Machine*. https://linux-kvm.org/page/Main_Page. 2007.

[23] Intel Corporation. *Intel Trust Domain Extensions (TDX)*. 2024. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html.

[24] François Costa and Charly Castes. *virt-sail: A Sail to Rust Compiler*. 2024. URL: https://github.com/CharlyCst/virt-sail.

[25] Victor Costan and Srinivas Devadas. “Intel SGX Explained.” In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 86.

[26] John Criswell et al. “Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems”. In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2007, pp. 351–366.

[27] Leila Delshadtehrani et al. “PHMon: A Programmable Hardware Monitor and Its Security Use Cases”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 807–824. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/delshadtehrani.

[28] REMS Project Developers. *Sail Rewrite System*. https://github.com/rems-project/sail/blob/284c4795a25723139443dedee1d178f68ddb304e/src/lib/rewrites.ml#L4422. Accessed: 2025-03-15. 2025.

[29] K. Elphinstone et al. “Towards a practical, verified kernel”. In: *11th HotOS*. May 2007, pp. 117–122.

[30] Paul Emmerich et al. *The Case for Writing Network Drivers in High-Level Programming Languages*. 2019. arXiv: 1909.06344 [cs.NI]. URL: https://arxiv.org/abs/1909.06344.

[31] Manuel Fähndrich et al. “Language Support for Fast and Reliable Message-Based Communication in Singularity OS”. In: *Proceedings of the 1st EuroSys Conference*. Apr. 2006, pp. 177–190.

[32] Erhu Feng et al. “Scalable Memory Protection in the PENGLAI Enclave”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 275–294. ISBN: 978-1-939133-22-9. URL: https://www.usenix.org/conference/osdi21/presentation/feng.

[33] Andrew Ferraiuolo et al. “Komodo: Using verification to disentangle secure-enclave hardware from software”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 287–305. ISBN: 9781450350853. DOI: 10.1145/3132747.3132782. URL: https://doi.org/10.1145/3132747.3132782.

[34] Ben Fiedler et al. “Putting out the hardware dumpster fire”. In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HOTOS ’23. Providence, RI, USA: Association for Computing Machinery, 2023, pp. 46–52. ISBN: 9798400701955. DOI: 10.1145/3593856.3595903. URL: https://doi.org/10.1145/3593856.3595903.

[35] Ben Fiedler et al. “Specifying the de-facto OS of a production SoC.” In: 2023, pp. 18–25.

[36] Trusted Firmware. *Trusted Firmware RMM*. 2024. URL: https://www.trustedfirmware.org/projects/tf-rmm/.

[37] Five Embedded Dev. *Hypervisor - RISC-V Privileged ISA Manual*. https://five-embeddev.com/riscv-priv-isa-manual/Priv-v1.12/hypervisor.html. Accessed: 2024-11-15. 2024.

[38] RISC-V Foundation. *RISC-V Supervisor Binary Interface (SBI) Specification*. https://github.com/riscv-non-isa/riscv-sbi-doc. Accessed: 2024-12-07. 2024.

[39] François. *print_string.asm*. Accessed: 2025-03-15. 2025. URL: https://github.com/francois141/Operating-system/blob/master/boot/print_string.asm.

[40] Tal Garfinkel et al. “Terra: A Virtual Machine-Based Platform for Trusted Computing”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2003.

[41] Ronghui Gu et al. “CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 653–669. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[42] M. Hohmuth and H. Tews. “The VFiasco approach for a verified operating system”. In: *2nd PLOS*. July 2005.

[43] Michael Hohmuth et al. “Reducing TCB Size by Using Untrusted Components—Small Kernels Versus Virtual-Machine Monitors”. In: *Proceedings of the 11th SIGOPS European Workshop*. Sept. 2004.

[44] IBM. *ACE Patch for Linux Kernel 6.3-rc4*. Accessed: 2025-02-17. 2023. URL: https://github.com/IBM/ACE-RISCV/blob/main/hypervisor/patches/linux/6.3-rc4/0002-ace.patch.

[45] IBM. *COVE Patch for Linux Kernel 6.3-rc4*. Accessed: 2025-02-17. 2023. URL: https://github.com/IBM/ACE-RISCV/blob/main/hypervisor/patches/linux/6.3-rc4/0001-cove.patch.

[46] Atalay Ileri et al. “Proving Confidentiality in a File System Using DiskSec”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 323–338.

[47] Information Assurance Directorate. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*. http://www.niap-ccevs.org/cc-scheme/pp/pp_skpp_hr_v1.03/. Version 1.03. June 2007.

[48] Intel. *Intel Software Guard Extensions (Intel SGX)*. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html. 2023.

[49] RISC-V International. *RISC-V Cryptographic Extension*. https://github.com/riscv/riscv-crypto. Accessed: 2024-12-01. 2024.

[50] RISC-V International. *RISC-V Vector Extension Specification*. https://github.com/riscvarchive/riscv-v-spec/blob/master/v-spec.adoc. Accessed: 2024-12-01. 2024.

[51] Daniel Jackson. “Lightweight Formal Methods.” In: 2001, p. 1.

[52] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. “Establishing Browser Security Guarantees through Formal Shim Verification”. In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 113–128. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/jang.

[53] Neelu S. Kalani et al. *Automatic ISA analysis for Secure Context Switching*. 2025. arXiv: 2502.06609 [cs.OS]. URL: https://arxiv.org/abs/2502.06609.

[54] Kaspersky. *LogoFAIL: UEFI Vulnerabilities Exploiting Image Parsing Flaws*. 2023. URL: https://www.kaspersky.com/blog/logofail-uefi-vulnerabilities/50160/.

[55] Frederic Khayat and Prof Edouard Bugnion. "Ecole Polytechnique Fédérale de Lausanne". In: (2025).

[56] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[57] Mark Kuhne, Stavros Volos, and Shweta Shinde. *Dorami: Privilege Separating Security Monitor on RISC-V TEEs*. 2024. arXiv: 2410.03653 [cs.CR]. URL: https://arxiv.org/abs/2410.03653.

[58] Ilia A. Lebedev et al. "Sanctorum: A lightweight security monitor for secure enclaves". In: *CoRR* abs/1812.10605 (2018). arXiv: 1812.10605. URL: http://arxiv.org/abs/1812.10605.

[59] Dirk Leinenbach and Thomas Santen. "Verifying the Microsoft Hyper-V Hypervisor with VCC". In: *Proceedings of the 2nd World Congress on Formal Methods*. FM '09. Eindhoven, The Netherlands: Springer-Verlag, 2009, pp. 806–809. ISBN: 9783642050886. DOI: 10.1007/978-3-642-05089-3_51. URL: https://doi.org/10.1007/978-3-642-05089-3_51.

[60] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2009, pp. 1–12. DOI: 10.1145/1480881.1480888. URL: https://doi.org/10.1145/1480881.1480888.

[61] H. Li et al. "ACRN: A Big Little Hypervisor for IoT Development". In: *Proceedings of the 15th International Conference on Virtual Execution Environments (VEE)*. 2019, pp. 31–44.

[62] Shih-Wei Li et al. "Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3953–3970. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei.

[63] Xiaoyong Li et al. "Design and Verification of the Arm Confidential Compute Architecture". In: *Proceedings of the 16th Symposium on Operating System Design and Implementation (OSDI)*. 2022, pp. 465–484.

[64] Jochen Liedtke. "Improving IPC by Kernel Design". In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. Dec. 1993, pp. 175–188. DOI: 10.1145/168619.168633.

[65] Jochen Liedtke. "Towards Real Microkernels". In: *Communications of the ACM* 39.9 (Sept. 1996), pp. 70–77. DOI: 10.1145/234215.234467.

[66] Canonical Ltd. *Ubuntu 24.04.1 RISC-V Preinstalled Server Image*. https://cdimage.ubuntu.com/releases/24.04/release/ubuntu-24.04.1-preinstalled-server-riscv64.img.xz. Repository: https://cdimage.ubuntu.com/. 2024.

[67] *MacOS Hypervisor.framework*. https://developer.apple.com/documentation/hypervisor.

[68] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. 2nd ed. Microsoft Press, 2004.

[69] *Memtier Benchmark*. https://github.com/RedisLabs/memtier_benchmark.

[70] Microsoft Security Response Center (MSRC). *CVE-2022-34302: UEFI Secure Boot Bypass Vulnerability*. Accessed: 2025-01-12. 2022. URL: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-34302.

[71] Microsoft Security Response Center (MSRC). *CVE-2022-34303: UEFI Secure Boot Bypass Vulnerability*. Accessed: 2025-01-12. 2022. URL: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-34303.

[72] L. Nelson et al. "Scaling symbolic evaluation for automated verification of systems code with Serval". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 2019, pp. 225–242.

[73] OASIS Virtual I/O Device (VIRTIO) TC. *Virtio Specification v1.2*. https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html. Accessed: 2024-03-15. 2023.

[74] Compiler Optimizations. *Hoisting*. 2024. URL: https://compileroptimizations.com/category/hoisting.htm.

[75] Wojciech Ozga et al. "Towards a Formally Verified Security Monitor for VM-based Confidential Computing". In: *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP2023. 2023.

[76] Pierluigi Paganini. *Three flaws allow attackers to bypass UEFI Secure Boot feature*. https://securityaffairs.com/134334/hacking/uefi-secure-boot-feature-flaw.html. 2022.

[77] *PKFAIL: Vulnerability in Supermicro BIOS firmware, July 2024*. https://www.supermicro.com/en/support/security_PKFAIL_Jul_2024.

[78] Gerald J. Popek and Robert P. Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[79] Fedora Project. *Fedora 40 RISC-V Rawhide Server Image for VisionFive2*. https://dl.fedoraproject.org/pub/alt/risc-v/disk_images/Fedora-40/VisionFive2/20240903.n.0/Fedora.riscv64-Rawhide_server_20240903.n.0.raw.zst. Repository: https://dl.fedoraproject.org/. 2024.

[80] REMS Project. *Sail C Backend*. Accessed: 2024-11-14. 2024. URL: https://github.com/rems-project/sail/tree/sail2/src/sail_c_backend.

[81] REMS Project. *Sail Coq Backend*. Accessed: 2024-11-14. 2024. URL: https://github.com/rems-project/sail/tree/sail2/src/sail_coq_backend.

[82] REMS Project. *Sail OCaml Backend*. Accessed: 2024-11-14. 2024. URL: https://github.com/rems-project/sail/tree/sail2/src/sail_ocaml_backend.

[83] RISC-V. *Sail RISC-V*. 2024. URL: https://github.com/riscv/sail-riscv.

[84] Timothy Roscoe et al. "Barrelfish: A Manycore Operating System for the Multicore Era". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2009, pp. 387–400. DOI: 10.1145/1629575.1629623.

[85] Ravi Sahita et al. "CoVE: Towards Confidential Computing on RISC-V Platforms". In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. CF '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 315–321. ISBN: 9798400701405. DOI: 10.1145/3587135.3592168. URL: https://doi.org/10.1145/3587135.3592168.

[86] Michael Sammler et al. "Islaris: verification of machine code against authoritative ISA semantics". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 825–840. ISBN: 9781450392655. DOI: 10.1145/3519939.3523434. URL: https://doi.org/10.1145/3519939.3523434.

[87] Oliver Schwarz and Mads Dam. "Formal verification of secure user mode device execution with DMA". In: *Proceedings of the 10th International Haifa Verification Conference (HVC 2014)*. Haifa, Israel, Nov. 2014, pp. 236–251.

[88] A. Seshadri et al. "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes". In: *16th SOSP*. Oct. 2007, pp. 335–350.

[89] Arvind Seshadri et al. "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes". In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Oct. 2007, pp. 335–350.

[90] AMD Sev-Snp. "Strengthening VM isolation with integrity protection and more". In: *White Paper, January* 53 (2020), pp. 1450–1465.

[91] Jonathan S. Shapiro, David F. Faber, and Jonathan M. Smith. "State Caching in the EROS Kernel—Implementing Efficient Orthogonal Persistence in a Pure Capability System". In: *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*. Nov. 1996, pp. 89–100.

[92] SiFive. *meta-sifive*. Accessed: 2025-02-08. 2025. URL: https://github.com/sifive/meta-sifive.

[93] Inc. SiFive. *EIC7700X Datasheet*. Accessed: 2025-02-07. 2025. URL: https://www.sifive.com/document-file/eic7700x-datasheet.

[94] Inc. SiFive. *HiFive Premier P550 Image Update Procedure*. Accessed: 2025-02-07. 2025. URL: https://www.sifive.com/document-file/hifive-premier-p550-image-update-procedure.

[95] Helgi Sigurbjarnarson et al. “Nickel: a framework for design and verification of information flow control systems”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 287–305. ISBN: 9781931971478.

[96] Helgi Sigurbjarnarson et al. “Push-Button Verification of File Systems via Crash Refinement”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 1–16.

[97] Lakshmi Singaravelu et al. “Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies”. In: *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys)*. Apr. 2006, pp. 161–174.

[98] Ltd. StarFive Technology Co. *VisionFive 2 Quick Start Guide*. Accessed: 2025-02-07. 2023. URL: https://doc-en.rvspace.org/VisionFive2/PDF/VisionFive2_QSG.pdf.

[99] *TDX module source code*. https://github.com/intel/tdx-module.

[100] Google Security Team. “AMD: Microcode Signature Verification Vulnerability”. In: (2025). Accessed: 2025-03-08. URL: https://github.com/google/security-research.

[101] Keystone Team. “Keystone: An Open Framework for Trusted Execution on Commodity Architectures”. In: *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS 2020)*. ACM, 2020, pp. 1–14. DOI: 10.1145/3319535.3372855.

[102] RISC-V ISA Development Team. *SSTC Extension Discussion*. Accessed: 2025-02-13. 2025. URL: https://groups.google.com/a/groups.riscv.org/g/isa-dev/c/8W4Sf-iqutY.

[103] RISC-V Software Development Team. *RISC-V ISA Simulator (Spike)*. https://github.com/riscv-software-src/riscv-isa-sim. Accessed: 2024-12-05. 2024.

[104] TechPowerUp. *Sinkclose Vulnerability Affects Every AMD CPU Dating Back to 2006*. Accessed: 2024-10-25. 2024. URL: https://www.techpowerup.com/325488/sinkclose-vulnerability-affects-every-amd-cpu-dating-back-to-2006 (visited on 10/25/2024).

[105] *The BSD Hypervisor*. https://bhyve.org/.

[106] H. Tuch, G. Klein, and G. Heiser. “OS verification — now!” In: *10th HotOS*. USENIX, June 2005, pp. 7–12.

[107] Harvey Tuch and Gerwin Klein. “Verifying the L4 virtual memory subsystem”. In: *Proceedings of the NICTA Formal Methods Workshop on OS Verification*. Sydney, Australia, Oct. 2004, pp. 73–97.

[108] Stanford University. *Rust: Memory Safety*. https://stanford-cs242.github.io/f18/lectures/05-1-rust-memory-safety.html. Accessed: 2024-11-15.

[109] Amit Vasudevan et al. “Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework”. In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 430–444. DOI: 10.1109/SP.2013.36.

[110] Amit Vasudevan et al. “überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor”. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 87–104. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/vasudevan.

[111] A. Velte and T. Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[112] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. “Scale and Performance in the Denali Isolation Kernel”. In: *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Dec. 2002.

[113] William Wulf et al. “HYDRA: The Kernel of a Multiprocessor Operating System”. In: *Communications of the ACM* 17 (1974), pp. 337–345. DOI: 10.1145/361011.361073.

[114] Yongwang Zhao and David Sanán. “Rely-Guarantee Reasoning About Concurrent Memory Management in Zephyr RTOS”. In: *Proceedings of the 31st International Conference on Computer-Aided Verification (CAV 2019)*. New York, NY, July 2019, pp. 515–533.

[115] Mo Zou et al. “Using Dynamically Layered Definite Releases for Verifying the RefFS File System”. In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 629–648. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/zou.

Appendix : Artifact evaluation

Miralis has an extra repository to reproduce the benchmarks. The Benchmark infrastructure is meant to be automatic and contains therefore very few commands. Please follow the instructions below to reproduce the results from this thesis. You also need to send a Miralis image on the VisionFive2 board and Premier P550 board. Please follow the instructions from this repositories to do so: *https://github.com/epfl-dcsl/visionfive2-doc/* and *https://github.com/epfl-dcsl/premierp550-doc*.

```
# Install git and download the repository containing the benchmarks
sudo apt-get install git

# Clone the repo on both the computer and the board
git clone https://github.com/epfl-dcsl/miralis-benchmark

# Get the ip of the board and write it in common.sh
ip a

# Install the softwares on the board
./install.sh

# Go to common.sh and add the corresponding IP

# Now run the workload from the computer with the correct argument
./run_workload.sh [board|miralis|protect|offload]

# Generate the plots on the computer
./generate_plots.sh
```

Listing 1: Instructions to reproduce the results