Kicking the Firmware Out of the TCB with the Miralis Virtual Firmware Monitor

Charly Castes
EPFL, Switzerland

Neelu S. Kalani
EPFL, Switzerland

Sofia Saltovskaia
EPFL, Switzerland

Noé Terrier
EPFL, Switzerland

Abel Vexina Wilkinson
EPFL, Switzerland

Edouard Bugnion
EPFL, Switzerland

Abstract

The role of firmware has evolved over the past decades. Not only is firmware responsible for discovering, initializing, and monitoring the system's chipset, board, and devices, but it also acts as the root of trust and plays a leading role in confidential computing. Yet vulnerabilities in the non-security critical part of the firmware have repeatedly led to the compromise of the core TCB of the system.

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 and verifiable *virtual firmware monitor*.

We present the design of Miralis, the first virtual firmware monitor. Miralis can successfully boot Linux with a virtualized OpenSBI on RISC-V. We demonstrate through construction that the M-mode of RISC-V architecture meets the Popek & Golberg criteria for classical virtualization. Our initial evaluation shows that Miralis removes vendor-provided, platform-specific firmware from the TCB with no measurable impact on boot and run-time performance.

ACM Reference Format:
Charly Castes, Neelu S. Kalani, Sofia Saltovskaia, Noé Terrier, Abel Vexina Wilkinson, and Edouard Bugnion. 2024. Kicking the Firmware Out of the TCB with the Miralis Virtual Firmware Monitor. In *2nd Workshop on Kernel Isolation, Safety and Verification (KISV '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3698576.3698764

1 Introduction

In a computer system, the firmware (*e.g.*, BIOS, UEFI, OpenSBI, *etc.*) is a low-level software running at the highest privilege level on the machine, such as ARM's EL3, x86_64's SMM, or RISC-V's M-mode. Historically, the firmware's role was to initialize the hardware at boot time and provide management services at run time. Specifically, the firmware initializes and interfaces with the board or SoC-specific devices, sensors, power controllers, *etc.* Thus, firmware closely interfaces with proprietary intellectual property and is often distributed as opaque binary blobs rather than an open-source software. For instance, developing firmware for Intel processors requires signing an NDA [2].

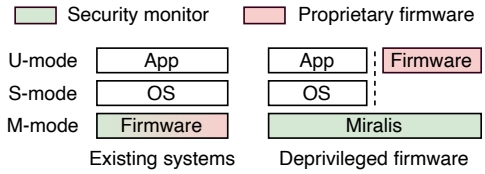

Figure 1. Comparison between existing systems, with collocated security monitor and firmware, and our proposed design with deprivileged firmware.

With the growing popularity of confidential computing [9, 21, 24, 25, 27, 29], the firmware plays an additional security-critical role in enforcing strict isolation policies to protect trusted execution environments (TEEs). The firmware has thus become a crucial part of the run-time TCB of the system.

Unfortunately, **these two roles of modern firmware are in tension with each other**. Enforcing isolation requires transparency, minimal code footprint, and correctness guarantees. On the other hand, initializing and managing vendor-specific hardware often leads to the firmware being shipped as an opaque binary blob comprising of large drivers, a large codebase, and a complex upgrade cycle. This has resulted in an unwelcome and significant increase in the size of the TCB and most of it is not security critical code.

As with any other complex software, firmware is also a source of recurring, exploitable vulnerabilities that can compromise the security of the entire system [3, 4, 6, 8, 13, 31, 34, 37]. The consequences of these vulnerabilities in the firmware include enabling secure boot bypass, system compromise due to continued use of leaked keys in outdated firmware hindered by complex firmware update processes, vulnerabilities that lead to leaking of secret keys, *etc.*

KISV '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1301-9/24/11…$15.00
https://doi.org/10.1145/3698576.3698764

Although firmware has become the most crucial software component of the TCB, little care has been taken to address the fact that a large part of the firmware is not security critical, and yet increases the attack surface significantly. This is a direct consequence of clinging on to the way firmware is architected traditionally, despite the evolution of the firmware's responsibilities. Security critical code is either added as part of existing firmware, such as OpenSBI and TrustedFirmware-A (TF-A) [27, 29], or executes at a lower privilege level [16]. In both cases **the security-critical code is co-located with, or controlled by the hardware-management code**, as depicted in Figure 1.

The problem of conflating distinct concerns within a single software component is a recurring issue in system design. It now also applies to firmware. The solution requires refactoring the software into components with enforced modularity and interfaces. For example, micro-kernels are now widely recognized for providing stronger security guarantees and used in the most security critical systems [22, 26, 30]. Similarly, hypervisors have undergone a transformation with the shift from type I [12, 39] to type II, the move toward user-space VMMs [7, 14, 20], and attempts to partition hypervisors even further [19, 28].

In this paper, we argue that firmware should undergo a similar transformation to keep up with the security standard expected from the confidential computing infrastructure. Rewriting existing firmware would incur a massive engineering effort and cost to adapt multiple layers of the stack. We instead propose a *backward-compatible* solution based on virtualization.

We remark that under some circumstances virtualization can be implemented efficiently and with a low complexity provided that the architecture meets the formal requirements from Popek & Goldberg (§3). We further observe that a complete virtualization of the architecture is not necessary if the only purpose is to virtualize firmware designed to run at the highest privilege level. We introduce the notion of *virtual firmware monitors* capable of running firmware within a virtual machine, with a strong implication on the threat model (§4). We leverage this observation to implement Miralis, a Rust-based RISC-V virtual firmware monitor that exposes a virtual M-mode (§5). We aim to prove the functional correctness of Miralis through a lightweight verification framework that compares symbolically our implementation with the RISC-V executable specification (§6). Finally, we evaluate a practical use-case by deploying Miralis on a VisionFive2 RISC-V board, running unmodified, sandboxed firmware images with minimal overhead (§7).

2 Motivation

Firmware is the software root of trust of the system. Yet firmware is not immune from bugs and is a very compelling target for either malicious hardware vendors or attackers potentially with stolen keys [3, 4, 6, 8, 13, 31, 34, 37].

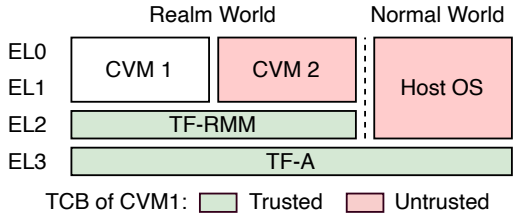

Figure 2. TCB of a confidential VM (CVM 1) with ARM CCA. CCA relies on a security monitor (TF-RMM) to isolate CVMs from each other, and on the firmware (TF-A) to isolate the Realm world from the host OS.

Trust in computer systems is derived from a series of measurements (cryptographic hashes) of successive software stages in the boot process. For instance, in a static root of trust process, each component verifies that the next software to execute in the boot process is a known valid and authorized version. Any malicious or exploitable software in the boot chain can bypass the verification of the next stages and ultimately compromise the system. Even with a *dynamic* root of trust measurement, part of the boot chain is verified via a static root of trust, and thus the firmware stays in the TCB [10].

Not only is firmware part of the root of trust chain, but it is also actively relied upon for providing the security guarantees of confidential computing. For the sake of example consider ARM CCA, depicted in Figure 2. Both research and industry emphasise the security properties of the EL2 software (TF-RMM), which is responsible for partitioning confidential memory among confidential VMs. Indeed, TF-RMM has undergone a thorough verification effort [29]. Yet, it is the EL3 firmware that is responsible for marking memory pages as confidential. While the example in Figure 2 is specific to CCA, TDX [24] and CoVE [36] follow a similar design and suffer from a similar TCB footprint.

Unfortunately, firmware has bugs, which can give attackers complete control over the system. Firmware is a *complex* software: it is responsible for the initialization, configuration, and monitoring of the system's chipset, board, and devices, and exposes a non-trivial interface to OSes and hypervisors; and it is responsible to provide security guarantees of confidential computing. The EDK II repository, an open-source UEFI SDK, has more than 1.6 million lines of code as of the May 2024 release [1] and more than 25 CVEs have already been registered so far. On ARM the EL3 interface specification is scattered across more than 9 documents (PSCI, SDEI, TrustZone, RMM, Management Mode, and more) and the interface is only expected to increase over time to provide backward compatibility.

Besides, firmware is a compelling target for malicious actors that either have access to the signing key or can carry out a supply chain attack, as compromising the firmware provides complete control of the system. The firmware update process requires the use of the vendor's private key to sign the firmware image. Unfortunately, even signed images cannot always be trusted. There are cases of attacks using a stolen key [3], and original manufacturers can push malicious updates either because they are forced to do so, or because of a compromised supply chain. Indeed, firmware relies on external libraries such as cryptography, compression, or network stack, any of which can be the target of sophisticated threat actors like the xz library was [5].

3 Background

Virtualization is a mature and proven concept across the industry. It offers three valuable, and yet often incompatible properties: strong isolation, full backward compatibility, and close to native performance. Those were first described as the *resource control*, *equivalence*, and *efficiency* properties in Popek & Goldberg's seminal 1974 paper on formal virtualization requirements [35].

In their paper, Popek & Goldberg describe a *sufficient* criterion for an instruction set architecture (ISA) to be virtualizable. The Popek & Goldberg's criteria states that the ISA is virtualizable if every *sensitive instruction* is a *privileged instruction*. A sensitive instruction is (loosely) defined as an instruction that either: (1) changes the configuration of the system, or (2) depends on the configuration of the system.

For instance, the RISC-V mret instruction is sensitive as it can transfer control across privilege levels, while add is not sensitive as its behavior is independent of the current privilege level and system registers (CSRs). For a precise definition of sensitive instructions, please refer to the original paper [35]. x86-32 is a famous example of a non-virtualizable architecture. The popf instruction, for instance, is one of the 17 instructions that violate the criteria: it is sensitive as it leaks the interrupt enable flag, but it is not privileged [15].

ISAs that satisfy the criteria are said to be *classically virtualizable*, they can be virtualized through a simple and efficient trap & emulate approach. ISAs that are *not* classically virtualizable can still be virtualized through pure software approaches, at the cost of greater complexity. The VMWare VMM, for instance, circumvents the non-virtualizability of x86-32 through selective use of dynamic binary translation [15].

4 Virtual firmware monitors

We propose a new class of systems called *virtual firmware monitors* (VFMs) whose purpose is to deprivilege firmware through virtualization. A VFM makes it possible to remove the non-security critical part of the firmware from the TCB. The VFM runs at the highest privilege level and its sole function is to enforce a security policy, *not* to manage hardware or provide backward compatibility. While this paper describes the basic functionalities necessary for deprivileging firmware, we envision that VFMs can be extended to provide additional security guarantees, such as isolating confidential VMs from untrusted firmware similar to how hypervisors can protect applications from an untrusted OS [17, 18, 23, 32, 40].

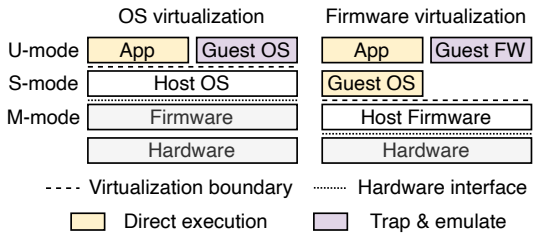

Figure 3. Comparison between traditional OS virtualization and our proposed solution for firmware virtualization.

4.1 Firmware virtualization

Virtualization traditionally targets the OS or hypervisor privilege levels, whereas a VFM virtualizes only the highest privilege level. Because the Popek & Goldberg criteria qualify instructions as either privileged or unprivileged, it implicitly draws a line in the privilege hierarchy. In 1974 when the paper was published machines usually had two privilege levels, but modern ISAs often have three of four. Privileged instructions are usually understood as non-user-level instructions. For instance, the RISC-V *privileged architecture* manual covers both S-mode and M-mode. Yet the Popek & Goldberg criteria can be interpreted with a more restricted set of privileged instructions, such as M-mode only.

Figure 3 contrasts traditional OS virtualization from our proposed solution for firmware virtualization. On RISC-V, this places the virtualization boundary between S-mode and M-mode rather than between U-mode and S-mode. An important property of the design is that the OS and hypervisor modes are *not* virtualized, meaning that: (1) OS and hypervisors execute at native speed, only firmware execution performance is affected, and (2) the VFM implementation is simplified because it does not need to emulate complex OS functionalities such as virtual memory.

4.2 Threat model

The role of the VFM is to deprivilege non-security critical firmware to remove it from the TCB. Virtualization guarantees that the VFM cannot be compromised even in the presence of adversarial firmware. For instance, the Keystone security monitor [27] could be implemented within a VFM rather than co-located with OpenSBI to additionally protect enclaves from untrusted firmware. Finally, side-channel, transient-execution attacks, and CPU bugs are out of scope but can be mitigated with additional hardening.

Table 1. RISC-V privileged instructions

Instructions	Description
ecall, ebreak	Trap/call into higher privilege mode
mret, sret	Return from trap
csrrw, csrrs, csrrc	CSR read/write, read/set, read/clear
csrrwi, csrrsi, csrrci	CSR operations with immediate
sfence.vma	Memory fence
wfi	Wait for interrupt

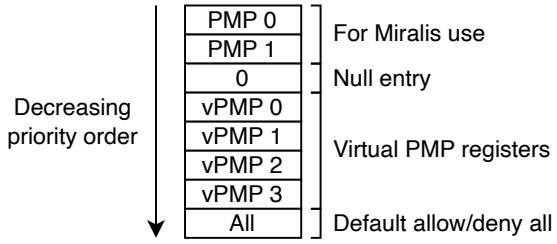

Figure 4. Multiplexing 8 physical PMP registers

5 M-mode virtualization with Miralis

To demonstrate the concept of VFM we designed and implemented Miralis, a RISC-V VFM. Miralis is an M-mode software that exposes a virtual M-mode (vM-mode) in which untrusted firmware can safely execute without the possibility to compromise Miralis, similar to how a hypervisor protects itself from untrusted VMs. In the rest of this section, we describe two key elements of the M-mode virtualization: CPU and memory protection virtualization.

5.1 CPU virtualization

Miralis has two operation modes: vM-mode for the firmware and direct execution for OS and applications. Firmware executing in vM-mode can trigger a transition to unmodified S or U-mode using the mret instruction. Whereas, when the OS traps into M-mode, Miralis automatically switches to vM-mode to forward the trap to the firmware. We refer to the switches between vM-mode and unmodified S or U-mode as *world switches*. Since traps are forwarded to and handled by the firmware, Miralis does not need to implement any firmware functionalities (*i.e.,* servicing SBI calls).

Execution in vM-mode is implemented as execution in physical U-mode (see Figure 3). Running virtual firmware in U-mode causes more traps to Miralis than S-mode would but is required for correctness: S-mode execution has access to sensitive instructions, such as changing the page table root, which do not trap to M-mode and therefore violate Popek & Goldberg's criteria. Indeed, M-mode firmware expects to be able to change the page table root but *not* to be affected by the change. However a de-privileged firmware running in S-mode would be affected by address translation, breaking the equivalence property RISC-V only implements a few privileged instructions (listed in Table 1) in the core specification, all of which can easily be emulated.

The difficulty of CPU emulation does not come from the instructions themselves, but from the control and status registers (CSRs). The specification defines over a hundred CSRs, each controlling or reporting about a specific aspect of the machine, yet not all are necessarily implemented or required by software. The current implementation of Miralis supports 49 CSRs and covers the needs of Linux, OpenSBI, and Zephyr (see §7). CSRs are managed by Miralis as a set of virtual registers that can be partitioned into two categories: the ones that modify the runtime behavior of M-mode software, and the ones that do not. CSRs that do not have any impact on M-mode execution can be read or written from the virtual CSRs directly, but accesses need to be filtered to enforce valid bit patterns. CSRs with side-effects on M-mode execution need special handling. For instance, the virtual mie (M-mode interrupt enabled) needs to be reflected in the physical mie for proper interrupt emulation. Fortunately there are only a few such registers, Miralis has support for 6 of them.

On world switch from vM-mode to OS execution, Miralis installs the virtual CSRs into the physical registers. All the S-mode CSRs as well as most M-mode CSRs can be installed directly. Some CSRs, however, need to stay under the control of Miralis. This is for instance the case of the PMP registers, as described in §5.2. On world switch from the OS to vM-mode the reverse operation happens: Miralis saves the physical CSRs (potentially updated by the OS) into the virtual registers and resumes firmware execution.

5.2 PMP virtualization

On RISC-V M-mode software does not use an MMU to manage and protect its own memory but relies on physical memory protection (PMP) instead. PMP virtualization does not require shadow or two-level page tables, removing the need for complex dynamic memory management.

PMP rules are specified with a set of up to 8 configuration and 64 addresses registers that can each protect one contiguous segment of memory. The rules are ordered by priority, the first matching entry determines the corresponding access rights. Memory segments can have different encodings, each with varying alignment and granularity constraints. The *Top of Range* (ToR) encoding in particular uses the previous entry to determine the start address of the segment, with the special case of zero when applied to the first PMP entry.

Miralis exposes virtual PMP to the firmware by multiplexing the physical registers, as pictured in Figure 4. The first (highest priority) entries are used by Miralis to protect its own memory and, for instance, MMIO regions. The next group of registers host the virtual PMP, with a null entry separating them from the first group. The null entry is required to properly emulate the ToR encoding behavior with virtual PMP. Finally, the last (lowest priority) entry is used to emulate the default access rights: by default, M-mode has access to all memory, while S-mode has access to none.

Table 2. Miralis lines of code decomposition.

	LoC	Percent
HW definitions & emulation	1892	55.8%
Assembly & Rust wrappers	563	16.6%
Control loop	346	10.2%
Configuration & debugging	278	8.2%
Other	313	9.2%
Total	3392	100%

6 Verifying virtual firmware monitors

The correctness of the VFM, the most privileged software on the machine, is crucial to the security of the overall system. Fortunately, VFMs offer unique opportunities in terms of transparency and formal verification. First because a VFM decouples security enforcement from proprietary hardware management and therefore can be made open-source like Intel's TDX module and ARM's TF-RMM. Second because VFMs (and hypervisors in general) expose a well-defined interface that is (1) formally defined, and (2) finite for the relevant part, simplifying automated formal verification. In the rest of this section, we describe a method for a fully automated, push-button approach to VFM verification.

The bulk of the complexity in VFMs is concentrated in the emulation of traps and privileged instructions. Table 2 shows lines of code repartition in Miralis, our Rust-based VFM, as counted by cloc. Emulation and hardware definitions constitute most of the code and can be error-prone when manually derived from the documentation. In Miralis, this accounts for more than 55% of the code.

We propose a method for automatic formal verification of the correctness of privileged instructions and trap emulation in VFMs by exhaustively comparing the hand-written emulation with the specification through symbolic execution. For ISAs such as ARM and RISC-V the specifications include a machine-readable format in addition to the English document used by developers. RISC-V uses Sail [11] as its machine-readable specification, from which reference emulators can be automatically generated (in C or OCaml). If for every possible instruction and register state the hand-written implementation produces the same new register state as the reference, then the hand-written implementation is correct. Exhaustive checking can be efficient with symbolic execution because the hardware interface is, by design, finite [33].

We demonstrated the feasibility of our proposed method by verifying the implementation of the mret instruction in Miralis and found a bug in our implementation in the process. First, we wrote a tool to convert Sail code into equivalent Rust. While Rust and Sail can be surprisingly similar on the surface there are semantic gaps that needed to be filled. For instance, Sail provides a bitvector primitive type that we model as a custom type in Rust. We then used Kani [38], a symbolic execution tool for Rust, to prove the equivalence. We executed the mret instruction against a set of virtual registers initialized with arbitrary symbolic values using both the Miralis implementation and the Rust code derived from the Sail specification and obtained two equal symbolic register states. At the time of writing our proof of concept, we require to manually patch the reference Rust implementation due to limitations in our Sail to Rust translator, which limits its applicability to more complex instructions such as CSR read and writes which generate over 2400 lines of reference Rust code. All current limitations can be lifted with a more robust and complete Sail to Rust translation.

7 Evaluation

Our current version of Miralis is implemented in 3392 lines of Rust code and supports two platforms: Qemu for development and the VisionFive2 board as a demonstrator. In the rest of this section, we describe our experiments running unmodified firmware and OS on top of Miralis, how we integrated Miralis in the VisionFive2 boot process, and our preliminary overhead analysis.

7.1 Running unmodified firmware and OS with Miralis

We demonstrated the feasibility of running unmodified OSes and firmware on top of a VFM by running a stock Linux kernel and Zephyr RTOS on top of Miralis. We ran both OSes on the Qemu virt board with a single core. Miralis is loaded as the entry point in Qemu using the `bios` option and the Linux and Zephyr images are pre-loaded in memory at known addresses. We embedded Linux directly as an OpenSBI payload, by building both Linux and OpenSBI with the default configuration and transferring control from Miralis to OpenSBI.

Booting Linux demonstrates the capability of Miralis to run an unmodified OS with virtualized firmware and exercises the world switch between vM-mode and direct OS execution. Zephyr is an RTOS for micro-controllers, it runs in M-mode only but uses a richer set of features than OpenSBI such as timers and interrupts to schedule multiple M-mode threads. By running Zephyr seamlessly in vM-mode we demonstrate the ability of Miralis to virtualize a wide range of firmware and its applicability in more constrained environments such as embedded systems.

7.2 Inserting Miralis in the VisionFive2 boot flow

We chose the VisionFive2 board as a demonstrator because of its support for Linux and non-trivial boot flow with multiple boot stages. The VisionFive2 board uses a 64 bits JH7110 RISC-V SoC with four U74 cores at 1.5 GHz and one smaller S7 core. In its current shape our prototype does not yet support multi-core, thus we restrict our experiments to running on a single U74 core.

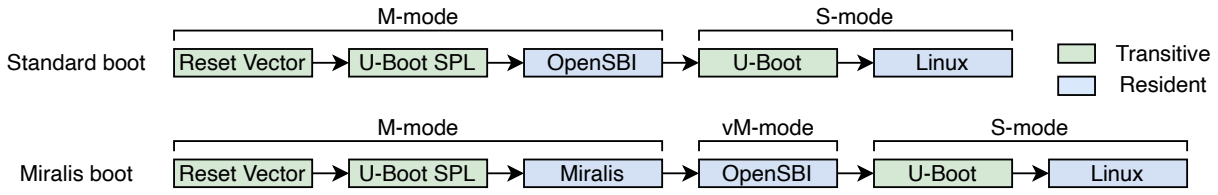

Figure 5. Boot flow

Table 3. Cost of trap & emulate and world switches on the VisionFive2.

	Instructions retired	Cycles
Trap & emulate	2854	3209 ± 200
World switch to OS	3293	3736 ± 23
World switch to firmware	3079	3459 ± 35

Table 4. Number of traps during Linux boot

Phase	Trap & emulate	World switches	Total exits
OpenSBI initialization	488	–	488
Linux boot	1252	206	1458
Total	1740	206	1946

The boot of the VisionFive2 relies on three firmware images: U-Boot SPL, U-Boot, and OpenSBI. Figure 5 depicts the standard boot flow of the VisionFive2 board. When powering-up the board the control is initially transferred to the reset vector, a small program in ROM that transfers control directly to the first firmware image: U-Boot SPL. U-Boot SPL is an M-mode software whose role is to initialize the hardware (such as DRAM) and load the next firmware stage. In the standard boot flow, OpenSBI is the next firmware to run and the first firmware to stay resident in memory until powering down. OpenSBI exposes the SBI interface to Linux with functions such as configuring timers and controlling power. Finally, OpenSBI loads U-Boot as an S-mode program which loads and jumps into Linux.

We demonstrated running an unmodified, deprivileged firmware by modifying the boot flow to run OpenSBI in vM-mode. Figure 5 depicts the boot flow when executing with Miralis. Miralis replaces OpenSBI as the first resident firmware, and only trusted M-mode firmware at runtime. The U-Boot image on the VisionFive2 board needs to be loaded at a specific address, but because Miralis does not use virtual memory to virtualize M-mode page tables are not an option. Instead, we used U-Boot SPL to load Miralis *after* the OpenSBI and U-Boot image to keep the same memory layout and remove the need for re-compiling U-Boot.

Our Miralis prototype boots unmodified OpenSBI and U-Boot on the VisionFive2 board. It currently does not yet support full initialization of Linux.

7.3 Overhead analysis

The overhead of Miralis comes from the extra cost of trapping and emulating privileged instructions during firmware execution, as well as increased transition cost to and from the firmware because of world switches. We conducted an initial evaluation of Miralis by running micro-benchmarks to evaluate the cost of firmware traps and world switches. We then measured the number of traps to Miralis during a Linux boot (where most of the traps to firmware happen) to estimate the overall impact of Miralis.

For micro-benchmarks, we execute 1000 traps or world-switches on the VisionFive2 board and report the number of instructions retired and cycles as read from the performance counters in Table 3. The number of instructions retired is deterministic in our experiments: Miralis only uses statically allocated data structures and has no sources of non-determinism other than interacting with UART. The cost of trap & emulate was estimated a write to the privileged mscratch register. The number of cycles can vary and depends on the CPU, we report numbers for the VisionFive2 U74 core (in-order, dual issue). We found that the cost of trap & emulate is low, typically between 3200 and 3800 cycles or at most 2.5μs at 1.5 GHz. Note that our micro-benchmarks do not reflect the cost of flushing TLBs due to permission changes in the PMP on world switches, which might impact the overall performance.

We then measured the number of traps to Miralis during a complete Linux boot on the Qemu virt board. Table 4 shows the number of exits for OpenSBI initialization until the first transition to S-mode, and the world-switches and exits for the rest of the boot (U-Boot and Linux). We measured a total of 1946 exists, or a 4.9 milliseconds overhead on boot time by extrapolating results from our micro-benchmarks.

8 Conclusion

Firmware is part of the TCB of all modern systems, and yet represents a significant threat and attack vector. We introduced the concept of *virtual firmware monitors* to remove the firmware from the TCB without breaking backward-compatibility through virtualization. We built Miralis, a virtual firmware monitor for RISC-V, and demonstrated running unmodified firmware binaries in a virtualized M-mode.

References

[1] Edk ii project. https://github.com/tianocore/edk2.git.

[2] Intel system bring-up toolkit. https://www.intel.com/content/www/us/en/developer/tools/oneapi/system-bring-up-toolkit.html.

[3] Pkfail: Vulnerability in supermicro bios firmware, july 2024. https://www.supermicro.com/en/support/security_PKFAIL_Jul_2024.

[4] Sinkclose vulnerability affects every amd cpu dating back to 2006. https://www.techpowerup.com/325488/sinkclose-vulnerability-affects-every-amd-cpu-dating-back-to-2006.

[5] Xz utils backdoor cve. https://nvd.nist.gov/vuln/detail/CVE-2024-3094.

[6] 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.

[7] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)* (2020), pp. 419–434.

[8] Alcorn, P. 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.

[9] AMD. Sev-snp: Strengthening vm isolation with integrity protection and more. *White Paper, January* (2020).

[10] ARM. Drtm architecture for arm.

[11] Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K. E., Norton, R. M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., and Sewell, P. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang. 3*, POPL (2019), 71:1–71:31.

[12] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[13] Behling, D. 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.

[14] Bellard, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC, FREENIX Track* (2005), pp. 41–46.

[15] Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., and Wang, E. Y. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst. 30*, 4 (2012), 12:1–12:51.

[16] Castes, C., and Baumann, A. Sharing is leaking: blocking transient-execution attacks with core-gapped confidential vms. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24)* (2024).

[17] Castes, C., Ghosn, A., Kalani, N. S., Qian, Y., Kogias, M., Payer, M., and Bugnion, E. Creating Trust by Abolishing Hierarchies. In *Proceedings of The 19th Workshop on Hot Topics in Operating Systems (HotOS-XIX)* (2023), pp. 231–238.

[18] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J. S., and Ports, D. R. K. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)* (2008), pp. 2–13.

[19] Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P. A., and Warfield, A. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (2011), pp. 189–202.

[20] Community, T. L. K. Linux kernel virtual machine. https://linux-kvm.org/page/Main_Page, 2007.

[21] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (2017), pp. 287–305.

[22] Hildebrand, D. An Architectural Overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures* (1992), pp. 113–126.

[23] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z., and Witchel, E. InkTag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)* (2013), pp. 265–278.

[24] Intel. Architecture specification: Intel trust domain extensions (intel tdx) module. https://software.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-module-1eas.pdf, 2023.

[25] Intel. Intel software guard extensions (intel sgx). https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2023.

[26] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D. A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 207–220.

[27] Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K., and Song, D. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the 2020 EuroSys Conference* (2020), pp. 38:1–38:16.

[28] Li, S.-W., Koh, J. S., and Nieh, J. Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium* (2019), pp. 1357–1374.

[29] Li, X., Li, X., Dall, C., Gu, R., Nieh, J., Sait, Y., and Stockwell, G. 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.

[30] Liedtke, J. On micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (1995), pp. 237–250.

[31] Lyons, J. It's official: Blacklotus malware can bypass secure boot on windows machines. https://www.theregister.com/2023/03/01/blacklotus_malware_eset/, 2023.

[32] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V. D., and Perrig, A. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy* (2010), pp. 143–158.

[33] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., and Wang, X. 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.

[34] Paganini, P. Three flaws allow attackers to bypass uefi secure boot feature. https://securityaffairs.com/134334/hacking/uefi-secure-boot-feature-flaw.html, 2022.

[35] Popek, G. J., and Goldberg, R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM 17*, 7 (1974), 412–421.

[36] Sahita, R., Shanbhogue, V., Bresticker, A., Khare, A., Patra, A., Ortiz, S., Reid, D., and Kanwal, R. CoVE: Towards Confidential Computing on RISC-V Platforms. In *Proceedings of the 20th ACM International Conference on Computing Frontiers (CF)* (2023), pp. 315–321.

[37] Smolár, M. When "secure" isn't secure at all: High-impact uefi vulnerabilities discovered in lenovo consumer laptops. https://www.welivesecurity.com/2022/04/19/when-secure-isnt-secure-uefi-vulnerabilities-lenovo-consumer-laptops/, 2022.

[38] VanHattum, A., Schwartz-Narbonne, D., Chong, N., and Sampson, A. Verifying Dynamic Trait Objects in Rust. In *ICSE (SEIP)* (2022), pp. 321–330.

[39] Waldspurger, C. A. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)* (2002).

[40] Zhang, F., Chen, J., Chen, H., and Zang, B. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (2011), pp. 203–216.