# Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode

Ronny Chevalier
HP Labs
ronny.chevalier@hp.com

Maugan Villatel
HP Labs
maugan.villatel@hp.com

David Plaquin
HP Labs
david.plaquin@hp.com

Guillaume Hiet
CentraleSupélec
guillaume.hiet@centralesupelec.fr

## ABSTRACT

Highly privileged software, such as firmware, is an attractive target for attackers. Thus, BIOS vendors use cryptographic signatures to ensure firmware integrity at boot time. Nevertheless, such protection does not prevent an attacker from exploiting vulnerabilities at runtime. To detect such attacks, we propose an event-based behavior monitoring approach that relies on an isolated co-processor. We instrument the code executed on the main CPU to send information about its behavior to the monitor. This information helps to resolve the semantic gap issue. Our approach does not depend on a specific model of the behavior nor on a specific target. We apply this approach to detect attacks targeting the System Management Mode (SMM), a highly privileged x86 execution mode executing firmware code at runtime. We model the behavior of SMM using invariants of its control-flow and relevant CPU registers (CR3 and SMBASE). We instrument two open-source firmware implementations: EDK II and coreboot. We evaluate the ability of our approach to detect state-of-the-art attacks and its runtime execution overhead by simulating an x86 system coupled with an ARM Cortex A5 co-processor. The results show that our solution detects intrusions from the state of the art, without any false positives, while remaining acceptable in terms of performance overhead in the context of the SMM (i.e., less than the 150 µs threshold defined by Intel).

## CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; **Systems security**; *Security in hardware*;

## KEYWORDS

Hardware-based monitoring, firmware, SMM, co-processor, CFI

**ACM Reference Format:**
Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. 2017. Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode. In *Proceedings of ACSAC 2017*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3134600.3134622

*ACSAC 2017, December 4–8, 2017, Orlando, FL, USA*


## 1 INTRODUCTION

Computers often relies on low-level software, like the kernel of an Operating System (OS) or software embedded in the hardware, called firmware. Due to their early execution and their direct access to the hardware, these low-level components are highly privileged programs. Hence, any alteration to their expected behavior, malicious or not, can have dramatic consequences on the confidentiality, integrity or availability of the system.

Boot firmware, like the Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) compliant firmware, is in charge of testing and initializing hardware components before transferring the execution to an OS. In addition to boot firmware, the platform initializes and executes runtime firmware code while the OS is running. On x86 systems, a highly privileged execution mode of the CPU, the System Management Mode (SMM) [41], executes runtime firmware code.

Any attacker that can change the original behavior of boot or runtime firmware, like skipping a verification step, can compromise the system. For this reason, tampering with the firmware is appealing for an attacker and sophisticated malware tries to infect it. Such malware is persistent, hard to detect, and does not depend on the OS installed on the platform [28, 33, 52].

Firmware code is stored on dedicated flash memory. On x86 systems, only runtime firmware code executed in SMM is allowed to modify the flash. It prevents a compromised OS from infecting the firmware. During the boot phase and before executing the OS, the boot firmware loads some code in System Management RAM (SMRAM). This code corresponds to privileged functions that will be executed in SMM. Then, the firmware locks the SMRAM and the flash (using hardware features) to prevent any modification from the OS. Furthermore, recent firmware uses cryptographic signatures during the boot process [30, 36, 67] and the update process [19] to ensure that only firmware signed by the vendor's key is updated and is executed. In addition, measurements (cryptographic hash) of all the components and configurations of the boot process can be computed and securely stored at boot time, to attest the integrity of the platform [34].

While cryptographic signatures and measurements provide code and data integrity at boot time, they do not prevent an attacker from exploiting a vulnerability in SMM at runtime [8, 27, 74]. Hence, we need ways to prevent vulnerabilities in SMM, or at least to detect intrusions exploiting such vulnerabilities.

Our work focuses on designing an event-based monitor for detecting intrusions that modify the expected behavior of the SMM

code at runtime. While monitoring the behavior of SMM is our primary goal, ensuring the integrity of the monitor itself is critical to prevent an attacker from evading detection. Thus, we isolate the monitor from the monitored component (i.e., the target) by using a co-processor.

A common issue affecting hardware-based approaches that rely on an isolated monitor is the semantic gap between the monitor and the target [48, 55, 63]. Such semantic gap issue occurs when the monitor only has a partial view of the target state. For example, if the monitor gets a snapshot of the physical memory without knowing virtual to physical mapping (e.g., CR3 register value on x86) it cannot reconstruct accurately the memory layout of the target. Our monitor addresses this issue by leveraging a communication channel that allows the target to send any information required to bridge this semantic gap. We enforce the communication of information relevant to the detection method via an instrumentation phase. In addition, we ensure that the attacker cannot forge messages without first being detected.

Our detection approach relies on a model of the expected behavior of the monitored component, while any significant deviation from this behavior is flagged as illegal. We chose an anomaly-based approach as we aim to detect exploits of unknown vulnerabilities.

In summary, our approach consists in detecting malicious behavior of a target program executed on a main CPU. The detection is implemented in a monitor executed on an isolated co-processor. We also instrument the target code to enforce the communication between the target and the monitor at runtime.

This approach can be applied to monitor various low level software, such as SMM or ARM TrustZone secure world [4], which have the following properties: expose primitives called infrequently by upper layers and perform minimal computation per primitive. Moreover, different detection approaches could be used. While generic, such approach introduces multiple challenges (e.g., the overhead involved by the communication, the provenance of the messages, or the integrity of the code added by the instrumentation phase). In this paper, we focus on the detection of attacks targeting the SMM code as a use case and show how we tackled these challenges. We enforce Control-Flow Integrity (CFI) [1, 14, 16, 59, 62, 71, 72, 80] and monitor the integrity of relevant CPU registers (CR3 and SMBASE) to illustrate the feasibility of our approach.

Our contributions are the following:

- We propose a new approach using an event-based monitor targeting low-level software (§ 2).
- We study the applicability of our approach using CFI to detect attacks against SMM runtime firmware code (§ 5).
- We develop a prototype implementing our approach.
- We evaluate our approach in terms of detection capability and performance overhead on real-world firmware widely used in the industry (§ 6).

This paper is structured as follows. First, in § 2, we provide an overview of our generic approach. Then, in § 3, we give a brief background on CFI and SMM. We detail the threat model associated with this use case in § 4. We describe the design and implementation of our prototype in § 5. In § 6, we evaluate our approach. In § 7, we compare our approach with related work. Finally, we conclude and propose some future work in § 8.

# 2 APPROACH OVERVIEW & REQUIREMENTS

In this section, we describe the generic concepts and requirements of our event-based behavior monitoring approach. As explained in § 1, such concepts could be used to monitor different targets and could rely on different detection approaches. We detail in § 5 one possible implementation of this generic approach to detect runtime attacks on SMM code using CFI.

Our approach, illustrated in Figure 1, relies on three key components, which we detail in the following subsections: a co-processor, a communication channel, and an instrumentation step. The co-processor isolates the monitor from the target. The target uses the communication channel to give more precise information about its behavior to the monitor. The instrumentation step enforces the communication.

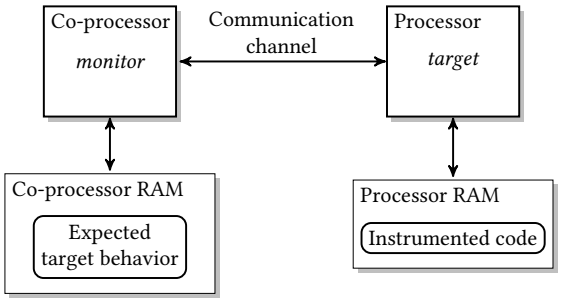



**Figure 1: High-level overview of the approach**

## 2.1 Co-processor

The integrity of the monitor is crucial, because it is a trusted component that we rely on to detect intrusions in our system. The monitor could also be used to start remediation strategies and restore the system to a safe state. If the attacker compromised our monitor, we could not trust the detection nor the remediation.

When the target and the monitor share the same resources (e.g., CPU or memory), it gives the attacker a wide attack surface. Thus, it is necessary to isolate the monitor from the target. Modern CPUs provide hardware isolation features (e.g., SMM or ARM TrustZone [4]) reducing the attack surface. However, if one wants to monitor the code executed in such environment, the monitor itself cannot benefit from these isolation features.

In our approach, we use a co-processor to execute the monitor. Such co-processor has its own execution environment and memory. Thus, the attacker cannot directly access this dedicated memory even if the target has been compromised. The attacker could only influence the behavior of the monitor via the communication channel, which becomes the only remaining attack surface. The simplicity of such an interface, however, makes it harder to find vulnerabilities and to attack the monitor. Such design reduces the attack surface.

In the following subsection, we discuss the requirements for our communication channel.

## 2.2 Communication with the monitor

Being isolated from the target, the monitor cannot retrieve entirely the execution context of the target. Thus, there is a semantic gap

between the current behavior of the target and what the monitor, executed on the co-processor, can infer about this behavior [7, 43]. For example, the monitor does not have sufficient information to infer the virtual to physical address mapping, nor the execution path taken at any point in time.

We introduce a communication channel between the monitor and the target. It allows the target to send messages to the monitor. Different types of information could be sent using this communication channel such as the content of a variable in memory, the content of a register, or the address of a variable. The nature of such information depends on the detection approaches implemented on the monitor, providing flexibility in our approach.

The communication channel is the only remaining attack vector against the monitor. Thus, how the monitor processes the messages and how the target sends them are an important part of the security of the approach. To this end, we require the following properties:

- **(CC1) Message integrity** If a message is sent to the monitor, it cannot be removed or modified. Otherwise, an attacker could compromise the target and then modify or delete the messages before they are processed by the monitor to hide the intrusion.
- **(CC2) Chronological order** Messages are retrieved by the monitor in the order of their emission. Otherwise, an attacker could rearrange the order to evade the detection.
- **(CC3) Exclusive access** The instrumented code has exclusive access to the communication channel. Otherwise, an attacker could forge messages faking a legitimate behavior.
- **(CC4) Low latency** Sending a message should be fast (e.g., sub-microsecond), because low-level components need to minimize the time spent performing their task to avoid impacting higher-level components and the user experience.

### 2.3 Instrumentation of the target

We enforce the communication from the target to the monitor by adding the communication code during an instrumentation step. This instrumentation step can be performed during the compilation or by rewriting the executable binary code.

Our approach relies on this enforcement, as should an attacker tamper with the instrumentation, the monitor would get inaccurate context of the behavior of the target making avoiding detection possible. Thus, the integrity of the instrumentation (i.e., the communication code of the target) is crucial. To this end, we require the following properties:

- **(I1) Boot time integrity** The code and data at boot time are genuine and cannot be tampered with by the attacker.
- **(I2) Runtime code integrity** The code cannot be modified by the attacker at runtime.

## 3 BACKGROUND

In this section, we provide an overview on control-flow hijacking and CFI. Then, we give some background regarding the SMM.

### 3.1 Control Flow Integrity (CFI)

Widely used defense mechanisms such as non-executable data and non-writable executable code impede attackers in their ability to exploit low-level vulnerabilities. Nevertheless, if an attacker managed to modify an instruction pointer due to a vulnerability, then program execution would be compromised. For example, in an x86 architecture, programs store the return address of function calls on the stack. An attacker could exploit a buffer overflow to overwrite the return address with an arbitrary one that redirect the execution flow. Code-reuse attacks, such as Return-Oriented Programming (ROP) [66] or Jump-Oriented Programming (JOP) [11, 17], use indirect branch instructions (i.e., indirect call to a function, return from a function and indirect jump) to chain together short instruction sequences of the existing code to perform arbitrary computations.

The enforcement of a policy over the control-flow can prevent such attack. This defense mechanism, called Control-Flow Integrity (CFI), enforces integrity properties for each indirect branch where the control-flow transfer is determined at runtime. It ensures that a given execution of a program follows only paths defined by a Control-Flow Graph (CFG). This graph represents all the legitimate paths that the program can follow. The CFG needs to be defined ahead of time and it can be computed via source code analysis [1], binary analysis [80], or execution profiling [76].

A typical way to enforce CFI is by instrumenting the code, e.g., during the compilation phase. This inlined-based approach adds runtime checks before each indirect branch [1, 71, 72]. If the address is not within a finite set of allowed targets, the program stops.

A fined-grained CFI combines a shadow call stack (i.e., an independent protected stack that only stores return addresses) and a precise CFG (i.e., a CFG with a small approximation regarding indirect branches) to enforce CFI on all indirect control transfers.

Some implementations [31, 80] sacrifice security over performance by building a less precise CFI. They either focus on protecting the backward-edge on the CFG (e.g., with a shadow call stack) or on protecting the forward-edge (e.g., indirect calls). Davi and Monrose [25] demonstrated that such implementations, called coarse-grained CFI, fail to protect against control-flow hijacking. Carlini et al. [16] also raised awareness on this issue by consolidating the argument that *without* stack integrity (i.e., without using a shadow call stack), CFI is insecure.

Our solution uses a type-based CFI inspired by the work of Niu and Tan [59] and Tice et al. [71]. We implement a shadow call stack and verify that each indirect call branches to a function with an expected type signature known at compile time (more details in § 5).

### 3.2 System Management Mode (SMM)

SMM [41] is a highly privileged execution mode of x86 processors. It provides the ability to implement OS-independent functions (e.g., advanced power management, secure firmware update, or configuration of UEFI secure boot variables) [41, 77]. The particularity of the SMM is that it provides a separate execution environment, invisible to the OS. The code and data used in SMM are stored in a hardware-protected memory region only accessible in SMM, called SMRAM. SMM is entered by generating a System Management Interrupt (SMI), which is a hardware interrupt. Software can also make the hardware trigger an SMI.

Access to the SMRAM depends on the configuration of the memory controller, done by the firmware during the boot. Once all the necessary code and data have been loaded in SMRAM, the firmware

locks the memory region so that it can only be accessed by code running in SMM, thus preventing an OS from accessing it. In addition, only the code executed in SMM can modify the firmware stored into flash to prevent malware, executing with kernel privileges, from overwriting the firmware and becoming persistent.

The particularity of an SMI is that it makes all the CPU cores enter SMM. It is non-maskable and non-reentrant. Hence, this interrupt must be processed as fast as possible, since the OS is paused during the handling of an SMI.

Despite hardware-based protection of the SMRAM, several attacks [8, 12, 20, 27, 60, 61, 65, 74, 75] were publicly disclosed. These attacks are proof-of-concepts that attackers could use to perform arbitrary code execution in SMM, once the SMRAM has been locked.

*Cache poisoning.* Two research teams [27, 74] independently discovered cache poisoning attacks in SMM. Since the cache is shared between all the execution modes of the CPU, the attack consists in marking the SMRAM region to be cacheable with a write-back strategy. Then, the attacker stores in the cache malicious instructions. After that, once an SMI is triggered, the processor fetches the instructions from the cache. Thus, the processor executes the malicious instructions of the attacker instead of the legitimate code stored in SMRAM. The solution is to separate the cache between non-SMM and SMM executions. This vulnerability has been fixed by adding a special-purpose register. Such register can only be modified in SMM and decides the cache strategy of the SMRAM.

*Insecure call.* Multiple firmware implementations [20] used call instructions to jump to code segments outside of the SMRAM. An attacker with kernel-level privileges can easily modify this code. These vulnerabilities have been fixed by forbidding the processor to execute instructions located outside of the SMRAM while in SMM.

Other vulnerabilities due to indirect calls [61, 75] allow attackers to perform code-reuse attacks against the SMM code. Such attacks are usually prevented by patching these vulnerabilities. Our approach can detect code-reuse attacks in general without requiring patching.

*Unchecked data.* Some SMI handlers rely on data provided by the OS (i.e., controlled by the attacker). If they do not sanitize such data, the attacker can influence the behavior of the SMM.

For example, pointer vulnerabilities in an SMI handler can lead to arbitrary write into SMRAM [8, 60, 65]. It can occur because the SMI handler writes data into a buffer located at an address controlled by the attacker. For example, such address can be provided thanks to a register that could have been modified by the attacker. Bulygin et al. [12] also demonstrated a similar attack by modifying the Base Address Registers (BAR) used to communicate with PCI devices.

It is the responsibility of SMI handlers to verify that the data given or controlled by the OS is valid. For example, they should check that the address of the communication buffer is not pointing into the SMRAM, and that the BARs point to valid addresses (i.e., not in RAM or SMRAM).

## 4 THREAT MODEL AND ASSUMPTIONS

As explained previously, the SMM is the last bastion of firmware security. It is the only mode that allows write access to the flash storage of the firmware, and its execution is invisible to the OS, thus a perfect place to hide malware [28]. In addition, it allows the attacker to perform actions that cannot be realized with kernel privileges. For example, if the attacker wants to remain persistent or modify security configurations (e.g., disable secure boot).

Every time a vulnerability related to the SMM has been reported it has been patched. Firmware, however, is not updated frequently [45]. Moreover, in practice, vendors typically use third-party code to build their firmware making code review and vulnerability management more difficult.

Hence, we assume that the attacker will find a vulnerability, but exploitation of such vulnerability implies a deviation from the expected behavior of the SMM code. Thus, our approach focuses on monitoring its behavior. Such anomaly-based approach is not limited to the detection of well-known attacks, but can also detect the exploitation of unreported (zero-day) vulnerabilities.

We assume that the code during the boot process is legitimate and that no attack is performed during that phase until the SMRAM is locked. Such an assumption is reasonable with the use of existing security mechanisms for recent firmware such as:

- An immutable hardware root of trust to verify that the boot firmware has a valid cryptographic signature from the vendor before its execution [36, 67],
- Cryptographic signatures during the update process [19],
- A Trusted Platform Module (TPM) chip to measure all the components of the boot process at boot time [34].

These mechanisms provide us with code and data integrity at boot time (I1, a requirement stated in § 2.3). In addition, since recent firmware use page tables [73] in SMM we can enable write protection [78, 79] and assume code integrity at runtime (I2).

Another key assumption is that the attacker cannot send messages in lieu of SMM without being detected. First, by design, messages cannot be sent by other components than the CPU and among the messages sent by the CPU only those sent in SMM are processed by the monitor (see § 5.3.2). Second, we assume that there is no vulnerability in SMM code that can be exploited by an attacker to forge messages without altering the control flow. Since any attempt to alter the control flow results in the emission of a message describing an invalid control flow (see § 5.4.1), the attacker cannot forge messages without first being detected.

Finally, we do not consider an attacker trying to impede the availability of the system (denial of service) by flooding the communication channel. The attacker already has sufficiently high privileges to perform a denial of service (e.g., shutdown the machine). We model such an attacker with the following capabilities:

- Complete control over the OS or the hypervisor, meaning that the attacker already found vulnerabilities that elevate its privileges to kernel-level or hypervisor-level,
- Complete control over the memory, except the SMRAM, which is protected,
- Cannot exploit hardware vulnerabilities (e.g., cache poisoning attacks [27, 74] or bypassing SMRAM protection),
- Can trigger as many SMIs as necessary,
- Can exploit a memory corruption issue in an SMI handler.

This threat model is close to those used in the different attacks described in § 3.2 (except for the cache poisoning attack).

# 5 SMM BEHAVIOR MONITORING

We apply our generic approach to monitor the behavior of the SMM code using CFI and by ensuring the integrity of relevant x86 CPU registers. The design of our solution is illustrated in Figure 2. In this figure, straight arrows represent the steps taken during runtime and dashed arrows the steps taken during the instrumentation phase (compilation time). We describe our implementation in more details in the following subsections.

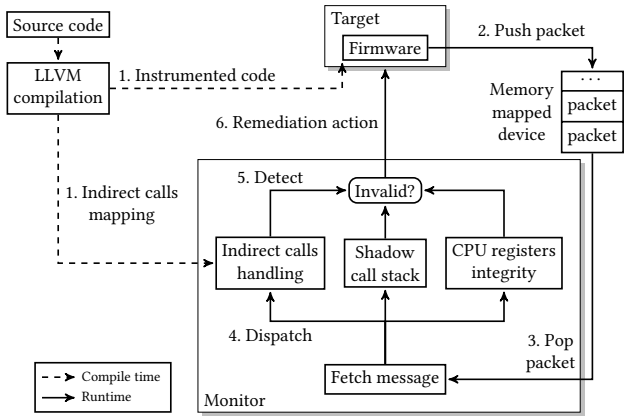



**Figure 2: High-level overview of the implementation**

## 5.1 Detection method

*5.1.1 CFI.* We enforce a CFI policy, because it is suited to detect attacks on low-level vulnerabilities that often appears in code written in C. Our monitor, executed on the co-processor, verifies that the control-flow information sent by the target is valid.

The monitor implements a type-based CFI inspired by the work of Niu and Tan [59] and Tice et al. [71]. It ensures that the address used in an indirect call matches the address of a function having an expected type signature known at compile time. For example, the call site `s->func(s, 1, "abc")` is an indirect call where `func` has `int (*func)(struct foo*, int, char *)` as a type signature. Thus, the monitor ensures that the address of `func` used at that call site always points to a function having the same signature. In addition, the monitor implements a shadow call stack to ensure the integrity of return addresses on the stack.

A type-based CFI over-approximates the set of expected pointers with all functions with the same type signature. In practice, type-based CFI gives small equivalence classes [14] where one equivalence class contains all the possible targets for one call site. An alternative could be to use a points-to analysis such as the work from Lattner et al. [47]. This type of analysis can sometimes give precise results (i.e., the complete set of pointers). However, in practice, as shown by Evans et al. [29], such analysis often fails to give the accurate set of pointers resulting in large equivalent classes such as all the available functions in the program.

Finally, our approach isolates the detection logic, the model of the behavior, and the data structures (e.g., shadow call stack and indirect call mappings) with the use of an isolated co-processor. It prevents attackers from tampering with it. Thus, we provide a more robust CFI using external monitoring in comparison to inlined-based CFI [1].

*5.1.2 CPU registers integrity.* In addition to a CFI policy, the monitor ensures the integrity of relevant x86 CPU registers in SMM. It stores expected values in its memory at boot time and verifies the values sent by the target at runtime.

When entering SMM, the main CPU stores its context in the save state area, and restores it when exiting [41]. The location of the SMRAM, called the SMBASE, is saved in the save state area. The processor uses the SMBASE every time an SMI is triggered to jump to the SMM entry point. Hence, it is possible for an SMI handler to modify the SMBASE in the save state area, and the next time an SMI is triggered, the processor will use the new SMBASE. Such behavior is genuine at boot time to relocate the SMRAM to another location in RAM. At runtime, however, there is no valid reason to do this. If an attacker manages to change the SMBASE, it results in arbitrary code execution when the next SMI is triggered. Therefore, the monitor ensures that the SMBASE value does not change between SMIs at runtime.

In addition, the monitor ensures the integrity of MMU-related registers, like CR3 (i.e., an x86 register holding the physical address of the page directory). Such register is an interesting target for attackers [43]. Thus, protecting its integrity is needed since recent firmware enable protected mode and use page tables [73, 78, 79]. These registers are reset at the beginning of each SMI with a value stored in memory. Such value is not supposed to change at runtime. If an attacker succeeds in modifying this value stored in memory, then the corresponding register is under the control of the attacker at the beginning of the next SMI.

## 5.2 Co-processor

We take inspiration from the AMD Secure Processor, also known as the Platform Security Processor (PSP) [3], and the Apple Secure Enclave Processor (SEP) [56]. Both are used as a security processor to perform sensitive tasks and handle sensitive data (e.g., cryptographic keys). In those solutions, the main CPU cannot directly access the memory of the co-processor. It only asks the co-processor to perform security-sensitive tasks via a communication channel.

The PSP is an ARM Cortex A5 and the SEP is an ARM Cortex A7. Such processors are similar, they are both 32 bit ARMv7 with in-order execution and 8-stage pipeline. The main difference is that the A5 is single-issue and the A7 is partial dual-issue.

In our implementation, we chose a similar design and we use an ARM Cortex A5 co-processor to execute our monitor. It gives us the isolation needed and enough processing power to process the messages for our use case.

We implemented our monitor with approximately 1300 lines of Rust [57], a safe system programming language.

## 5.3 Communication channel

In this subsection, we look at how existing co-processors communicate with the main CPU and explain why they do not fit our requirements. Then, we describe how we design our communication mechanism to fulfill the properties we defined in § 2.2.

*5.3.1 Existing mechanisms.* A major characteristic of the communication channel is its performance, especially its latency, as each message sent impacts the overall latency of SMI handlers.

The Intel BIOS Test Suite (BITS) defined the acceptable latency of an SMI to 150 µs [40]. Delgado and Karavanic [26] showed that, if the latency exceeds this threshold, it causes a degradation of performance (I/O throughput or CPU time) or user experience (e.g., severe drop in frame rates in game engines).

Both the PSP and the SEP use mailbox communication channels to send and receive messages with the main CPU [2, 56]. Mailboxes work as follows. One processor writes to a mailbox register, which triggers an interrupt in a second CPU. Upon receiving the interrupt, the second CPU executes code that fetches the value in the mailbox, processes the message, and then writes a response.

We could use such a mechanism to fulfill our security properties (CC1 and CC2) by making the SMM code wait until the co-processor acknowledged the message. Shelton [68] studied the latency of mailboxes on Linux and measured on average a 7500 cycles latency. For example, with a 2 GHz clock this gives 3.75 µs per message. Thus, not fulfilling the low-latency requirement (CC4).

Since the mechanism used by existing co-processors, like the PSP or the SEP, does not allow low latency communication while fulfilling our security requirements, we designed a specific hardware component to that end.

*5.3.2 Restricted FIFO.* We designed a restricted First In First Out (FIFO) queue between the main CPU and the co-processor. This FIFO is implemented as an additional hardware component connected to the main CPU and the co-processor, because we want to re-use existing processors without modifying them.

The goal of the FIFO is to store the messages sent by the target awaiting to be processed by the co-processor. The FIFO only allows the main CPU to push messages and the co-processor to pop them. The FIFO receives messages fragmented in packets. Only our FIFO handles the storage of the messages, the attacker does not have access to its memory, thus it cannot violate the integrity of the messages. We consider single-threaded access to the FIFO, since only one core handles the SMI, while other cores must wait [41].[1]

We are using a co-processor with less processing power than the main CPU and the monitor usually processes messages at a lower rate than their production. Thus, the FIFO could overflow. Such a case would happen if the monitored component would be continuously executing, which is not the case with SMM code. Most of the time the main CPU will execute code in kernel land or userland, which are not monitored and hence do not send any message. An SMI, on the other hand, will create a burst of messages when triggered. Hence, the only case where the FIFO could overflow is if an attacker deliberately triggered SMIs at very high rate, which would be detected as an attack.

We use a fast interconnect between the main CPU executing the monitored component and the FIFO. The precise interconnect depends on the CPU manufacturer. In the x86 world two major interconnects exist: QuickPath Interconnect (QPI) [39] from Intel and HyperTransport [35] from AMD.

These interconnects are used for inter-core or inter-processor communication and are specifically designed for low latency. For example, CPU manufacturers are using them to maintain cache coherency. Furthermore, they have been leveraged to perform CPU-to-device communication [32, 53, 54]. The co-processor could be connected to the FIFO using these interconnects (using glue logic) or an interconnect with similar performance (e.g., AMBA [6]).

Our monitored component has a mapping between a physical address and the hardware component (i.e., the FIFO) allowing it to send packets via the interconnect. Routing tables are used by interconnects. Such routing tables are configured via a software interface (with kernel privileges) to decide where the packets are sent. Thus, as explained by Song et al. [69], it would be possible for an attacker to modify the routing tables to prevent the delivery of the messages to the FIFO. Such attack would be the premise of an attack against a vulnerable SMI handler. Therefore, at the beginning of each SMI, we enforce the mapping by overwriting the routing table in the SMM code to prevent such an attack.

In addition, the FIFO filters the messages by checking the SMI-ACT# signal of the CPU specifying whether the main CPU is in SMM or not [35, 41]. Hence, the monitor only processes messages sent in SMM and prevents an attacker from sending messages when the target is not executing (e.g., an attacker sending messages in kernel mode).

To summarize, this design fulfills the message integrity property (CC1), since the target can only push messages to the restricted FIFO. Moreover, if the queue is full it does not wrap over and the target enforces the routing table mapping. It fulfills the chronological order property (CC2), because it is a FIFO and there is no concurrent access to it while in SMM. In addition, it fulfills the exclusive access property (CC3), since we filter messages to ensure they only come from the SMM, the integrity of the instrumentation code is ensured with the use of page tables with write-protection enabled, and the attacker cannot forge messages without first being detected. Finally, we fulfill the last property (CC4) by using a low latency interconnect between the main CPU and the FIFO.

## 5.4 Instrumentation

The instrumentation step of our implementation that modifies the SMM code is twofold: (1) an instrumentation to send CFI related information; and (2) an instrumentation to send information regarding x86 specific variables.

As previously stated, the goal of the instrumentation is to *send* information to the monitor. In comparison to other approaches where they inline some verifications in the instrumented code, we only use `mov` instructions to send packets to our FIFO.

*5.4.1 CFI.* We rely on LLVM 3.9 [46], a compilation framework widely used in the industry and the research community, to instrument the SMM code. We implement two LLVM passes with approximately 600 lines of C++ code.

The first pass enforces the backward-edge CFI (i.e., a shadow call stack). It instruments the SMM code to send one message at the prologue and epilogue of each function. Such message contains the return address stored on the stack.

[1] At the beginning of each SMI, there is a synchronization code ensuring that only one core executes in SMM. This implies that we do not instrument the code responsible for the synchronization between the cores. Such code does not interact with any attacker-controlled data and cannot be influenced by the attacker, hence we trust it.

The second pass enforces forward-edge CFI (i.e., indirect calls always branch to valid targets). For each indirect call site, we assign a unique identifier (CSID), we create a mapping between their CSID and the type signature of the function called, and we add this type into a set of types called indirectly (SIND). We instrument each indirect call site to send the CSID and the branch target address to the monitor before executing the indirect call.

Then, for each function whose type signature is in SIND, we add a mapping between the function offset in memory and its type. This mapping gives us all the functions that could be called indirectly with their type signature and offset in memory.

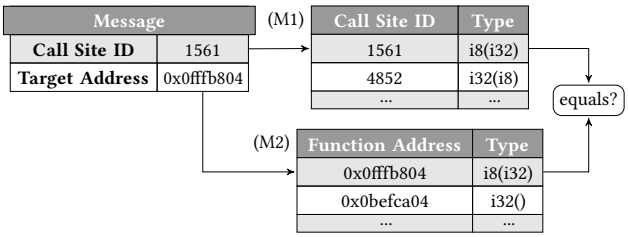



**Figure 3: Mappings used to verify indirect calls messages**

At the end of the build process, we provide two pieces of information: (1) a mapping between a CSID and a type; and (2) a mapping between an *offset* and the type of the function at that location. However, such information is not enough for the monitor to have the mapping at runtime. It only has the functions offset and not their final address in memory, hence the monitor needs the base address of the code used for the SMM. We provide this information to the monitor by instrumenting the firmware code to send the address during the initialization phase (before the SMRAM is locked). This way, at boot time, the monitor computes the final mapping by adding the offset to the corresponding base address.

Finally, as illustrated in Figure 3, the monitor can compute two mappings: (M1) a mapping between a CSID and its expected type; and (M2) a mapping between the address of a function and its type. Thanks to this information, the monitor can verify that the target address received in a message has the expected type according to the call site ID from the same message. The attacker can control the target address, but not the call site ID.

*5.4.2 CPU registers integrity.* We also instrument the SMM code to send some values related to x86 CPU registers. These values, such as SMBASE or the saved value of CR3, could be modified by an attacker to take control of the SMM or evade detection.

First, we add some code executed at boot time to send the current values to the monitor. Since there is no legitimate modification of these values at runtime, the monitor registers them. Secondly, we add some code executed at runtime to send the values at the end of each SMI.

# 6 EVALUATION

We evaluated our approach on two real-world implementations of code running in SMM. We first conducted a security evaluation of our approach using QEMU, as described in § 6.2. Then, we used the gem5 simulator to evaluate the runtime overhead of our approach, as detailed in § 6.3.

We used a simulation-based prototype in order to have enough flexibility in exploring the hardware architecture, in a manner that would have been difficult to achieve using real hardware, such as FPGA-based solutions.[2] A simulation allows us to simulate an interconnect and to simulate the delay it takes for the main CPU to send one packet to the restricted FIFO.

## 6.1 Experimental setup

We used EDK II [70] and coreboot [21], two real-world implementations of code running in SMM. EDK II is an open source UEFI compliant firmware used as the foundation for most vendor-based firmware. Coreboot is an open source firmware performing hardware initialization before executing a payload (e.g., legacy BIOS or UEFI compliant firmware). We built this firmware using our LLVM toolchain and we only instrumented the SMM related code.

*6.1.1 Simulator and emulator.* We both used a simulator and an emulator to validate our approach. The main goal of emulators is to be as feature-compatible as possible. However, they are not cycle-accurate and does not try to model accurately the performance of x86 or ARM platforms. Simulators, on the other hand, try to model accurately the performance of the platforms they simulate, but often do not implement all their features (e.g., no possibility to lock the SMRAM). Therefore, we use emulators to have all the SMM features, which is mandatory for security evaluation, and simulators to model accurately the performance of our implementation.

For the security evaluation, we used the QEMU 2.5.1 [9] emulator. We modified QEMU to emulate our communication channel.

We used the gem5 [10] cycle-accurate simulator to estimate the performance impact both on the main CPU by modeling an x86 system, and on the co-processor by modeling an ARM Cortex A5. Butko et al. [15] evaluated that gem5 gave a performance prediction with a 20% error on average.

We modified gem5 to simulate our FIFO communication channel. It allowed us to specify the delay (in nanoseconds) it takes to send or receive information from it. We give the parameters used for gem5 in Appendix A.

*6.1.2 Simulated communication channel delay.* We relied on previous studies on interconnects [18, 53] to estimate the delay of the communication channel.

Litz et al. [53] encountered a latency between 36 to 64 cycles to send one packet with HyperTransport on a CPU-FPGA platform.[3] Even with a small clock rate, for example 500 MHz, we can expect a latency of around 72 to 128 ns, close to an uncached memory access. Choi et al. [18] have similar results with QPI-based platforms.[4]

Hence, we simulated a delay of 128 ns to send one packet. This corresponds to the worst-case scenario to send one packet. Since the reference latency we have for AMD HyperTransport and Intel

[2] At the time of writing, to the best of our knowledge, there is no off-the-shelf FPGA-based solutions with direct access to HyperTransport or Intel QPI commercially available.

[3] Litz et al. [53] designed an FPGA card with the HTX3 interface, which is needed for point-to-point communication with HyperTransport. Xilinx used to sell such products but they are now discontinued.

[4] Choi et al. [18] had access to a QPI-based CPU-FPGA platform thanks to a collaboration between Intel and academics at that time.

QPI are for FPGA prototypes, lower latencies are expected with an ASIC implementation.

Furthermore, since we use a point-to-point connection, we did not consider a fluctuation of the latency. Moreover, as explained in § 5.3.2, only one core of the main CPU is running while in SMM.

Finally, we simulate the same interconnect and delay between the main CPU and the FIFO, and between the co-processor and the FIFO.[5]

*6.1.3 SMI handlers.* For our performance evaluation, we used SMI handlers from EDK II and coreboot. EDK II does not implement any hardware initialization nor vendor-related SMI handlers. At the time of writing, most of SMI handlers available in EDK II at runtime are dependent on hardware components that cannot be easily simulated (e.g., an Opal device or a TPM chip).

In our evaluation, we used the VariableSmm SMI handlers from EDK II. They manage variables within the SMM [77] thanks to four different handlers: GetVariable, SetVariable, QueryVariableInfo and GetNextVariableName (GNVN).

Since coreboot provides hardware initialization and vendor-related SMI handlers, we use them for our evaluation. In addition, these handlers communicate with devices, which can be simulated with gem5. A majority of these handlers are simpler compared to the VariableSmm SMI handlers. We used three SMI handlers for the Intel ICH4 i82801gx[6] and two for the AMD Agesa Hudson southbridge.[7] These SMI handlers process hardware events such as pressing the power button (PM1), General Purpose Events (GPE), Advanced Power Management Control (APMC) events, or Total Cost of Ownership (TCO) events.

## 6.2 Security evaluation

There is no public dataset of vulnerable SMM code, in contrast to userland applications. Attacks targeting the SMM are highly specific to the architecture and to the proprietary code of the platform. Such code is therefore not publicly available and would not execute on our experimental setup, thus cannot be used to test our solution.

Consequently, we have implemented SMI handlers with vulnerabilities similar to previously disclosed ones (see § 3.2) affecting real-world firmware. We reproduced attacks exploiting the following vulnerabilities giving arbitrary execution: (1) A buffer overflow in a SMI handler allowing an attacker to modify the return address stored on the stack [44]; (2) An arbitrary write allowing an attacker to modify a function pointer used in an indirect call [60]; (3) An arbitrary write allowing an attacker to modify the SMBASE [65]; and (4) An insecure indirect call where the function pointer is retrieved from a data structure controlled by the attacker [61].

As shown in Table 1, the monitor detected all these attacks as soon as it received and processed the messages, since these attacks modify the control-flow of the SMM code (i.e., its behavior). We did not encounter false positives, which is expected since we use a conservative strategy regarding indirect calls. Also, while bad software engineering practices using function type cast could introduce false positives, we did not encounter such case in the code we evaluated, as no function cast was present.

| Vulnerability | Attack Target | Security Advisories | Detected |
|---|---|---|---|
| Buffer Overflow | Return address | CVE-2013-3582 [22] | Yes |
| Arbitrary write | Function pointer | CVE-2016-8103 [23] | Yes |
| Arbitrary write | SMBASE | LEN-4710 [50] | Yes |
| Insecure call | Function pointer | LEN-8324 [51] | Yes |

**Table 1: Effectiveness of our approach against state-of-the-art attacks**

While our implementation detects these intrusions, an attacker could theoretically bypass our solution. First, by managing to send multiple forged packets without any other legitimate packets being sent in the middle. Second, doing so without redirecting the control-flow to send these forged packets (an attack out of our threat model, see § 4).

Finally, our CFI implementation performs a sound analysis to recover the potential targets of an indirect call. Therefore, the analysis is not complete and it would be possible for an attacker to redirect the control flow to a function that should have never been called, but that has the expected type signature. Nonetheless, we argue that a type-based CFI increases the difficulty for the attacker, since the only available targets for an indirect call are a subset of the existing functions within the SMRAM with the right type signature. Our analysis with EDK II gave 158 equivalence classes of size 1, 24 of size 2, 42 of size 3, 2 of size 5, 1 of size 9, and 1 of size 13. As mentioned by Burow et al. [14], a high number of small equivalence classes provides a precise CFG. A way to improve the precision of the CFG would be to combine our static analysis (providing some context-sensitivity with the type information), with a points-to analysis, such as the work from Lattner et al. [47]. Such points-to analysis can sometimes give the complete set of the functions being called at an indirect call site. An idea would be that if the points-to analysis gives a complete set, the monitor uses this information to validate an indirect call, otherwise it uses the over-approximation of the type signature.

## 6.3 Performance evaluation

As explained in § 5.3.1, the time spent in SMM is limited (threshold of 150 µs) [26, 40]. On that account, we evaluated the running time overhead of our solution on SMI handlers for the main CPU. We also evaluated the time it takes for the co-processor to process the messages sent by different SMI handlers. Thus, we can estimate the time between an intrusion, its detection, and remediation.

Finally, the size of firmware code is limited by the amount of flash (e.g., 8MB or 16MB). Thus, we evaluated the size of the firmware before and after our instrumentation.

*6.3.1 Runtime overhead.* The additional SMM code added with our instrumentation introduces two costs: the raw communication delay between the main CPU and the hardware FIFO; and the instrumentation overhead. The former is related to the time it takes the main CPU to push the packets to the FIFO. The latter is due to multiple factors, such as fetching and executing new instructions

[5] In practice, one would need to use a similar interconnect or a glue logic for the ARM architecture.
[6] E.g., present on motherboards from Apple, ASUS, GIGABYTE, Intel or Lenovo.
[7] E.g., present on motherboards from AMD, ASUS, HP, Lenovo or MSI.

or storing intermediate values resulting in register spilling (e.g., the return address of a function fetched from the stack).

We performed 100 executions of each SMI handler we selected for our evaluation (see § 6.1.3). For each SMI handler, we measured the time it takes for the original handler to execute, the cost of the communication, and the additional instrumentation overhead. The results we obtained are illustrated by Figure 4.

We see that even with a low latency of 128 ns there is a high overhead. It is due to the number of messages related to the shadow stack (see Table 2), while the number of messages for indirect calls or the integrity of the relevant CPU registers (SMBASE and CR3) are negligible. However, this overhead is below the 150 µs threshold [40] ensuring that the impact on the performance of the system is low and not noticeable for the user.

| | Number of packets sent | | | |
|---|---|---|---|---|
| SMI Handler | Shadow stack (SS) | Indirect call (IC) | SMBASE & CR3 (SC) | Total number of packets |
| **EDK II** | | | | |
| VariableSmm | | | | |
| SetVariable | 384 | 4 | 4 | 392 |
| GetVariable | 240 | 4 | 4 | 248 |
| QueryVariableInfo | 299 | 4 | 4 | 208 |
| GetNextVariableName | 212 | 4 | 4 | 220 |
| **coreboot** | | | | |
| Intel i82801gx | | | | |
| APMC/TCO/PM1 | 8 | 2 | 4 | 14 |
| AMD Agesa Hudson | | | | |
| APMC/GPE | 4 | 0 | 4 | 8 |

**Table 2: Number of packets sent during one SMI handler (Number of packets per message type: SS=2, IC=2, SC=4)**

*6.3.2 Co-processor performance.* We measured the time it takes for the monitor to process all the messages generated by one execution of each SMI handler. We made an average of 1000 executions. Results are illustrated in Figure 5.

For each SMI handler there is at least a factor of 4 between the time it takes for the target to execute the instrumented SMI handler and the time it takes for the co-processor to process all the messages that have been sent by the instrumented SMI handler. For example, we see in Figure 4 that it takes around 52 µs to execute the SetVariable SMI handler, and in Figure 5 that it takes around 230 µs to process all the messages. This means that there is a delay between an intrusion and its detection, but such delay will be less than a millisecond. Hence, the co-processor could start a remediation action within one millisecond after an intrusion occurred.

In our threat model, the attacker already has kernel privileges before attacking the SMM code. Our objective is not to detect intrusions that could have been done solely with kernel privileges, such as leaking confidential data. We consider that the final objective of the attacker is to remain persistent in the system even in the case of a reboot. In this case, the remediation action does not have to be taken immediately. This remediation would not prevent the intrusion, but will recover to a safe state.

*6.3.3 Firmware size.* For EDK II, our instrumentation added 17408 bytes to the firmware code. However, firmware is compressed before being stored in the flash and only a subset of the firmware is related to the SMM. We measured a 0.6% increase in size of the compressed firmware. Thus, our instrumentation incurs an acceptable overhead in terms of size for the firmware.

For coreboot, our instrumentation added 568 bytes for the AMD Agesa Hudson SMI handlers and 3448 bytes for the Intel i82801gx SMI handlers. However, we were not able to measure the whole firmware size when building coreboot with our LLVM toolchain, since coreboot does not support clang as a compiler.[8] We built separately the SMI handlers from coreboot toolchain for our evaluation, but compiling the whole firmware (not just the SMM related code) is not possible.

## 7 RELATED WORK

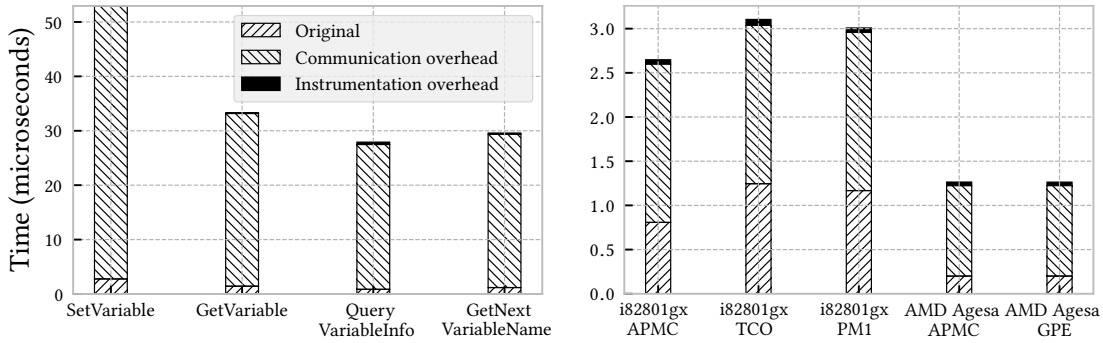
| | Approach | | | | | | |
|---|---|---|---|---|---|---|---|
| Property | Our approach | Copilot [63] | DeepWatch [13] | HyperSentry [7] | KI-Mon [48] | MGuard [55] | Hardware CFI [42, 49] |
| Can monitor SMM code | ● | ○ | ● | ○ | ● | ● | ● |
| Flexible | ● | ● | ● | ● | ● | ● | ○ |
| No semantic gap issue | ● | ○ | ○ | ● | ○ | ○ | ● |
| Detect transient attacks | ● | ○ | ○ | ○ | ● | ● | ● |
| No new/modified hardware | ○ | ○ | ● | ● | ○ | ○ | ○ |

**Table 3: Comparison between our work and the related work. ● : has the property ○ : does not have the property**

In this section, we discuss existing approaches to monitor low-level components such as firmware and kernels, using a hardware-based approach. To the best of our knowledge, the only commercially available technology that offers SMM integrity monitoring is HP Sure Start [37, 38]. It uses the chipset and the CPU to monitor SMM code integrity and relies on additional hardware to take actions per a predefined policy. The details of its implementation, however, are not public. Thus, we cannot compare it in details to other approaches in the literature.

The other approaches presented are not necessarily focused on monitoring the SMM (i.e., the firmware), but they could be adapted to that aim. We can distinguish two different types of approaches: snapshot-based approaches are presented in § 7.1 and event-based approaches in § 7.2. We summarize the comparison between these approaches and our work in Table 3.

[8] We could measure the size of coreboot compiled with gcc, but the size varies when using clang or gcc.




(a) EDK II VariableSmm (b) coreboot

**Figure 4: Time (in microseconds) to execute each SMI handler (averaged over 100 executions) with the original time, and the overhead divided between the communication overhead due to pushing packets to the FIFO and the instrumentation overhead.**

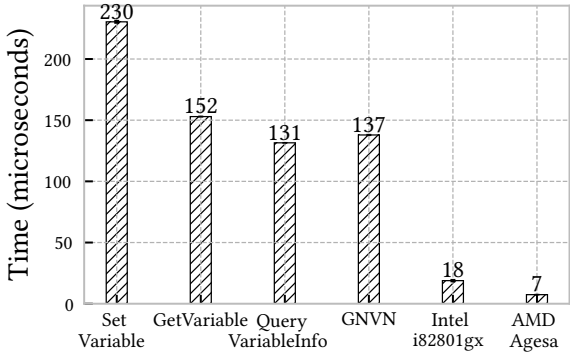



**Figure 5: Time (in microseconds) to process all the messages sent by one execution of each SMI handler for the co-processor**

## 7.1 Snapshot-based approach

The first approach consists in taking periodic snapshots of all or any part of the target state and then to analyze these snapshots to detect intrusions.

To the best of our knowledge, Zhang et al. [81] were the first to propose a co-processor for intrusion detection using a snapshot-based approach. However, they did not implement their design.

Notable implementations of such approach are Copilot [63], DeepWatch [13], and HyperSentry [7]. Copilot is a kernel integrity monitor using a co-processor on a PCI card to take periodic snapshots of the main memory. The authors also described how to write rules describing the relationships between kernel objects to detect the presence of kernel rootkits [64]. Copilot, however, cannot monitor the SMM since it does not have access to SMRAM. DeepWatch uses a similar approach to Copilot, but the monitor runs on an embedded core in the chipset, which allows the monitoring of the SMM. HyperSentry leverages the SMM to perform measurements giving access to the CPU-context, but it impedes its ability to monitor the SMM itself.

In general, these snapshot-based solutions are unable to detect transient attacks, where an attacker does not make persistent changes (e.g., one could erase its traces before each snapshot).

## 7.2 Event-driven approach

All the following event-driven approaches, like our approach, require a new specific hardware component or a modification of an existing hardware component.

Vigilare [58] snoops the memory bus traffic of the host by using an external hardware component to detect modifications of immutable regions of a kernel. This approach does not suffer from transient attacks: as soon as an illegal modification is made it is detected. KI-Mon [48], its successor, also monitors mutable kernel objects. MGuard [55] follows a similar approach, but incorporates the integrity monitor inside a DRAM DIMM device. One limitation, however, that affects these solutions, is their inability to access the CPU state of the host they are monitoring (semantic gap issue). Jang et al. [43] demonstrated the practicability of an evasion scheme by modifying the CR3 register. Our solution, while being event-driven, is not vulnerable to the CR3 attack since we monitor it as explained in § 5.4.2.

While our generic approach does not focus on CFI, our evaluation used CFI as a detection method to demonstrate the applicability of our approach. Thus, we compare our solution with hardware-based CFI approaches.

Lee et al. [49] use a co-processor and debugging features available in ARM processors to enforce CFI on the main CPU. Davi et al. [24] extend the instruction set of the processor to enforce a similar policy. A recent document from Intel [42] suggests that future Intel processors will have a backward-compatible CFI technology in hardware (and available for the SMM).

Hardware-based CFI approaches modify the processor or use additional hardware solely to enforce CFI. Our approach, on the other hand, is more flexible since different detection approaches could be implemented without modifying our hardware component. The flexibility of the solution is important, because the types of vulnerabilities exploited evolves over time.

As discussed in § 3, CFI can be implemented without hardware modifications by inlining the detection logic inside the target. Our approach, however, isolates the critical parts of the detection process such as the shadow call stack and the indirect call mappings.

## 8 CONCLUSION AND FUTURE WORK

In this work, we propose a new event-based approach for low-level software using three key components: a co-processor to isolate the monitor, a communication channel to reduce the semantic gap, and an instrumentation of the software to enforce the communication. We show that this approach can be followed to detect intrusions targeting SMM code by using CFI and by ensuring the integrity of relevant CPU registers (CR3 and SMBASE). However, it can implement different detection methods. Unlike other approaches, we solve the challenges of the semantic gap and the transient attacks while remaining flexible.

We implemented our approach by instrumenting and monitoring real-world firmware. The results show that we detect state-of-the-art attacks against the SMM, while remaining below the 150 µs threshold, thus avoiding any noticeable impact on the user.

For future work, we would like to investigate how we could leverage such a co-processor-based monitor to (1) start remediation strategies and study the impact on the user experience; and (2) apply our approach to monitor other targets and use other detection methods. For example, our approach could be used to monitor ARM TrustZone secure world [4], since it offers a similar environment than SMM (e.g., a non-secure bit to know whether the CPU is in the secure world or not, like the SMIACT# signal).

## A GEM5

| Parameter | | x86 | ARM Cortex A5 |
|---|---|---|---|
| CPU Type | | DerivO3Cpu | timing[i] |
| Clock | | 2 GHz | 500 MHz |
| Restricted FIFO latency | | 128 ns | 128 ns |
| Cache line size | | 32 B | 32 B |
| L1 I | Size | 32 KB | 16 KB[ii] |
| | Associativity | 2 | 2 |
| L1 D | Size | 64 KB | 16 KB[ii] |
| | Associativity | 2 | 4 |
| L2 | Size | 2 MB | 512 KB |
| | Associativity | 8 | 8 |
| DRAM | Type | DDR3_1600 | LPDDR3_1600_x32[iii] |
| | Size | 1024 MB | 10 MB |

[i] We use the timing model since the A5 is a single-issue in-order CPU and our evaluation mainly depends on load/store operations. [ii] The cache size has a range of options: 4 KB, 8 KB, 16 KB, 32 KB or 64 KB. [iii] Educated guess, based on the fact that this is a standard for low power consumption memory.

**Table 4: Parameters used with gem5 for the x86 and the ARM simulation**

In Table 4, we show the different parameters used to configure gem5 for the simulation of the main CPU and the co-processor. We use the default parameters of the out-of-order x86 simulation, except the CPU clock, which we set to a higher frequency. For the ARM Cortex simulation, we derived the parameters from the ARM technical reference manual [5].

## ACKNOWLEDGMENTS

The authors would like to thank and acknowledge the contribution of the following people (in alphabetical order) for their helpful comments, technical discussions, feedback and proofing of earlier versions of this paper: Vali Ali, Boris Balacheff, Pierre Belgarric, Rick Bramley, Chris Dalton, Carey Huscroft and Jeff Jeansonne. In addition, we would like to thank the anonymous reviewers for their feedback.

## REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS '05)*. ACM, Alexandria, VA, USA, 340–353.
[2] AMD. 2016. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 30h-3Fh Processors.* Advanced Micro Devices, Inc.
[3] AMD TATS BIOS Development Group. 2013. AMD Security and Server innovation. (March 2013). UEFI PlugFest.
[4] ARM. 2009. *ARM Security Technology: Building a Secure System using TrustZone Technology.* ARM.
[5] ARM. 2016. *ARM Cortex-A5 Technical Reference Manual.* ARM.
[6] ARM. 2017. AMBA Specifications. (2017). Retrieved September 10th, 2017 from https://www.arm.com/products/system-ip/amba-specifications
[7] Ahmed M Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C Skalsky. 2010. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, Chicago, IL, USA, 38–49.
[8] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, and Mickey Shkatov. 2015. A new class of vulnerabilities in SMI handlers. (2015). CanSecWest, Vancouver, Canada.
[9] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Anaheim, CA, USA, 41–46.
[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
[11] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*. ACM, Hong Kong, China, 30–40.
[12] Yuriy Bulygin, Oleksandr Bazhaniuk, Andrew Furtak, John Loucaides, and Mikhail Gorobets. 2017. BARing the System: New vulnerabilities in Coreboot & UEFI based systems. (2017). REcon Brussels.
[13] Yuriy Bulygin and David Samyde. 2008. Chipset based approach to detect virtualization malware. (2008). Black Hat USA.
[14] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.
[15] Anastasiia Butko, Florent Bruguier, Abdoulaye Gamatié, Gilles Sassatelli, David Novo, Lionel Torres, and Michel Robert. 2016. Full-System Simulation of big.LITTLE Multicore Architecture for Performance and Energy Exploration. In *Proceedings of the 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE Computer Society, 201–208.
[16] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium (SEC'15)*. USENIX Association, Washington, D.C., USA, 161–176.
[17] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, Chicago, IL, USA, 559–572.
[18] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, Austin, TX, USA, Article 109, 6 pages.
[19] David Cooper, William Polk, Andrew Regenscheid, and Murugiah Souppaya. 2011. BIOS protection guidelines. *NIST Special Publication* 800 (2011), 147.

[20] core collapse. 2009. ASUS Eee PC and other series: BIOS SMM privilege escalation vulnerabilities. (Aug. 2009). Retrieved January 26, 2017 from http://www.securityfocus.com/archive/1/505590

[21] The coreboot community. 2017. coreboot. (2017). Retrieved February 27, 2017 from https://www.coreboot.org/

[22] CVE-2013-3582 2013. CVE-2013-3582. (May 2013). Retrieved June 1st, 2017 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3582

[23] CVE-2016-8103 2016. CVE-2016-8103. (Sept. 2016). Retrieved June 1st, 2017 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8103

[24] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. 2015. HAFIX: Hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 74.

[25] Lucas Davi and Fabian Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, USA, 401–416.

[26] Brian Delgado and Karen L Karavanic. 2013. Performance implications of System Management Mode. In *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 163–173.

[27] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. 2009. Getting into the SMRAM: SMM Reloaded. (2009). CanSecWest, Vancouver, Canada.

[28] Shawn Embleton, Sherri Sparks, and Cliff C Zou. 2013. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks* 6, 12 (2013), 1590–1605.

[29] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, Denver, CO, USA, 901–913.

[30] UEFI Forum. 2016. *UEFI Platform Initialization Specification*. Version 1.5.

[31] Ivan Fratrić. 2012. *ROPGuard: Runtime prevention of return-oriented programming attacks*. Technical Report.

[32] Holger Fröning, Mondrian Nüssle, Heiner Litz, Christian Leber, and Ulrich Brüning. 2013. On achieving high message rates. In *Proceedings of the 13th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE Computer Society, Los Alamitos, CA, USA, 498–505.

[33] Sean Gallagher. 2013. Your USB cable, the spy: Inside the NSA's catalog of surveillance magic. Ars Technica. (31 Dec. 2013). Retrieved March 1, 2017 from https://arstechnica.com/information-technology/2013/12/inside-the-nsas-leaked-catalog-of-surveillance-magic/

[34] Trusted Computing Group. 2011. *TPM Main, Part 1 Design Principles*. Trusted Computing Group.

[35] Brian Holden, Don Anderson, Jay Trodden, and Maryanne Daves. 2008. *HyperTransport 3.1 Interconnect Technology*. MindShare Press.

[36] HP Inc. 2016. *HP Sure Start: Automatic Firmware Intrusion Detection and Repair System*. Technical Report. HP Inc. http://h10032.www1.hp.com/ctg/Manual/c05163901

[37] HP Inc. 2017. *HP Sure Start Gen3*. Technical Report. HP Inc. http://www8.hp.com/h20195/v2/GetPDF.aspx/4AA6-9339ENW.pdf

[38] HP Inc. 2017. *HP Sure Start with Runtime Intrusion Detection*. Technical Report. HP Inc. http://www8.hp.com/h20195/v2/GetPDF.aspx/4AA6-9340ENW.pdf

[39] Intel Corporation. 2009. Introduction to the Intel Quickpath Interconnect. (June 2009).

[40] Intel Corporation. 2011. bits-365. (March 2011). Retrieved January 26, 2017 from https://biosbits.org/news/bits-365/

[41] Intel Corporation. 2015. System Management Mode. In *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Chapter 34.

[42] Intel Corporation. 2016. Control-flow Enforcement Technology Preview. (June 2016).

[43] Daehee Jang, Hojoon Lee, Minsu Kim, Daehyeok Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2014. ATRA: Address Translation Redirection Attack against Hardware-based External Monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Scottsdale, AZ, USA, 167–178.

[44] Corey Kallenberg, John Butterworth, Xeno Kovah, and C Cornwell. 2013. Defeating Signed BIOS Enforcement. (2013). EkoParty, Buenos Aires.

[45] Xeno Kovah and Corey Kallenberg. 2015. How Many Million BIOSes Would you Like to Infect? (2015). CanSecWest.

[46] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, San Jose, CA, USA, 75–88. http://llvm.org/

[47] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. San Diego, California.

[48] Hojoon Lee, Hyungon Moon, Daehee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. 2013. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *Proceedings of the 22th USENIX Security Symposium*. USENIX Association, Washington, D.C., USA, 511–526.

[49] Yongje Lee, Ingoo Heo, Dongil Hwang, Kyungmin Kim, and Yunheung Paek. 2015. Towards a Practical Solution to Detect Code Reuse Attacks on ARM Mobile Devices. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy (HASP '15)*. ACM, Portland, OR, USA, Article 3, 8 pages.

[50] LEN-4710 2016. Lenovo Security Advisory: LEN-4710. (Sept. 2016). Retrieved June 1st, 2017 from https://support.lenovo.com/us/en/product_security/len_4710

[51] LEN-8324 2016. Lenovo Security Advisory: LEN-8324. (Nov. 2016). Retrieved June 1st, 2017 from https://support.lenovo.com/us/en/solutions/len-8324

[52] Philippe Lin. 2013. Hacking Team Uses UEFI BIOS Rootkit to Keep RCS 9 Agent in Target Systems. TrendLabs Security Intelligence Blog. (13 July 2013). Retrieved May 5, 2017 from https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-uses-uefi-bios-rootkit-to-keep-rcs-9-agent-in-target-systems/

[53] Heiner Litz, Holger Froening, Mondrian Nuessle, and Ulrich Bruening. 2008. VELO: A novel communication engine for ultra-low latency message transfers. In *Proceedings of the 37th International Conference on Parallel Processing*. IEEE Computer Society, 238–245.

[54] Heiner Litz, Maximilian Thuermer, and Ulrich Bruening. 2010. TCCluster: A Cluster Architecture Utilizing the Processor Host Interface as a Network Interconnect. In *Proceedings of the International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 9–18.

[55] Ziyi Liu, JongHyuk Lee, Junyuan Zeng, Yuanfeng Wen, Zhiqiang Lin, and Weidong Shi. 2013. CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, Tel-Aviv, Israel, 392–403.

[56] Tarjei Mandt, Mathew Solnik, and David Wang. 2016. Demystifying the Secure Enclave Processor. In *Black Hat Las Vegas*.

[57] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. ACM, Portland, OR, USA, 103–104. https://www.rust-lang.org/

[58] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. 2012. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*. ACM, Vienna, Austria, 28–37.

[59] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. *ACM SIGPLAN Notices* 49, 6 (2014), 577–587.

[60] Dmytro Oleksiuk. 2016. Exploiting AMI Aptio firmware on example of Intel NUC. (October 2016). Retrieved May 19, 2017 from http://blog.cr4.sh/2016/10/exploiting-ami-aptio-firmware.html

[61] Dmytro Oleksiuk. 2016. Exploring and exploiting Lenovo firmware secrets. (June 2016). Retrieved January 30, 2017 from http://blog.cr4.sh/2016/06/exploring-and-exploiting-lenovo.html

[62] PaX Team. 2015. RAP: RIP ROP. (Oct. 2015). H2HC.

[63] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. 2004. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, San Diego, CA, USA, 179–194.

[64] Nick L Petroni Jr, Timothy Fraser, AAron Walters, and William A Arbaugh. 2006. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th USENIX Security Symposium*. USENIX Association, Vancouver, B.C., Canada.

[65] Bruno Pujos. 2016. SMM unchecked pointer vulnerability. (May 2016). Retrieved May 19, 2017 from http://esec-lab.sogeti.com/posts/2016/05/30/smm-unchecked-pointer-vulnerability.html

[66] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012), 2.

[67] Xiaoyu Ruan. 2014. Boot with Integrity, or Don't Boot. In *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, Berkeley, CA, USA, Chapter 6, 143–163.

[68] Benjamin H Shelton. 2013. *Popcorn Linux: enabling efficient inter-core communication in a Linux-based multikernel operating system*. Master's thesis. Virginia Polytechnic Institute and State University.

[69] WonJun Song, John Kim, Jae-Wook Lee, and Dennis Abts. 2014. Security vulnerability in processor-interconnect router design. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, Scottsdale, AZ, USA, 358–368.

[70] Tianocore. 2017. EDK II. (2017). Retrieved January 26, 2017 from http://www.tianocore.org/edk2/

[71] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in gcc & llvm. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, USA, 941–955.

[72] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP '10)*. IEEE Computer Society, Oakland, CA, USA, 380–395.

[73] Dick Wilkins. 2015. UEFI Firmware – Securing SMM. (May 2015). Retrieved January 26, 2017 from http://www.uefi.org/sites/default/files/resources/UEFI_Plugfest_May_2015%20Firmware%20-%20Securing%20SMM.pdf

[74] Rafal Wojtczuk and Joanna Rutkowska. 2009. Attacking SMM memory via Intel CPU cache poisoning. (March 2009). Invisible Things Lab.

[75] Rafal Wojtczuk and Alexander Tereshkin. 2009. Attacking Intel BIOS. (July 2009). Black Hat USA.

[76] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*. IEEE Computer Society, Washington, D.C., USA, 1–12.

[77] Jiewen Yao, Vincent Zimmer, and Star Zeng. 2014. *A Tour Beyond BIOS Implementing UEFI Authenticated Variables in SMM with EDKII*. Technical Report. Intel.

[78] Jiewen Yao and Vincent J Zimmer. 2015. *A Tour Beyond BIOS Supporting an SMM Resource Monitor using the EFI Developer Kit II*. Technical Report. Intel.

[79] Jiewen Yao, Vincent J Zimmer, and Matt Flemming. 2015. *A Tour Beyond BIOS Memory Practices in UEFI*. Technical Report. Intel.

[80] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium*. USENIX Association, Washington, D.C., USA, 337–352.

[81] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. 2002. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop (EW 10)*. ACM, Saint-Émilion, France, 239–242.