SpecCert: Specifying and Verifying Hardware-based Security Enforcement

Thomas Letan, Pierre Chifflier, Guillaume Hiet, Pierre Néron, Benjamin Morin

SpecCert: Specifying and Verifying Hardware-based Security Enforcement

Thomas Letan[1], Pierre Chifflier[1], Guillaume Hiet[2], Pierre Néron[1] and Benjamin Morin[1]

[1] French Network Information Security Agency (ANSSI)
[2] CentraleSupélec

Abstract. Over time, hardware designs have constantly grown in complexity and modern platforms involve multiple interconnected hardware components. During the last decade, several vulnerability disclosures have proven that trust in hardware can be misplaced. In this article, we give a formal definition of *Hardware-based Security Enforcement* (HSE) mechanisms, a class of security enforcement mechanisms such that a software component relies on the underlying hardware platform to enforce a security policy. We then model a subset of a x86-based hardware platform specifications and we prove the soundness of a realistic HSE mechanism within this model using Coq, a proof assistant system.

1 Introduction

Modern hardware architectures have grown in complexity. They now are made of numerous devices which expose multiple programmable functions. In this article, we identify a class of security enforcement mechanisms we call Hardware-based Security Enforcement (HSE) such that a set of software components configures the hardware in a way which prevents the other software components to break a security policy. For instance, when an operating system uses the ring levels and memory paging features of x86 microprocessors to isolate the userland applications, it implements a HSE mechanism. A HSE mechanism is sound when it succeeds in enforcing a security policy. It requires (1) the hardware functions to provide the expected properties and (2) the software components to make a correct use of these hardware functions. In practice, both requirements are hard to meet.

First, hardware architectures comprise multiple interconnected devices which interact together. From a security perspective, it implies considering the devices both individually and as a whole. Hardware functions are not immune to security vulnerabilities. For instance, early versions of the `sinit` instruction implementation of the Intel TXT technology [13] allowed an attacker to perform a privilege escalation [21]. The legitimate use of a hardware mechanism can also break the security promised by another. For instance, until 2008, the x86 cache allowed to circumvent an access control mechanism exposed by the memory controller [17,22]. Secondly, hardware architectures have grown in complexity and,

as a consequence, HSE mechanisms too. There are many examples of security vulnerabilities which are the consequence of an incorrect HSE mechanism implementation [5,26,9].

In this paper, we introduce SpecCert, a framework for specifying and verifying HSE mechanisms against hardware architecture models. SpecCert relies on a three-steps methodology. First, we model the hardware architecture specifications. Then we specify the software requirements that must be satisfied by the trusted software components which implement the HSE mechanism. Finally, we prove that the HSE mechanism is sound under the assumption that the software components complies to the specified requirements. This implies the hardware involved in the HSE mechanism indeed provides the security properties they promise. We believe this approach to be beneficial to both hardware designers and software developers. The former can verify their hardware mechanism assumptions and the latter can get a formal specification to implement the HSE mechanism.

In Section 2, we give a formal definition of the SpecCert formalism. In Section 3, we define a model of x86-based hardware architectures to verify HSE mechanisms targeting software isolation policies using publicly available Intel specifications. In Section 4, we verify the soundness of the HSE mechanism implemented in many x86 computer firmware codes to isolate the code executed while the CPU is in System Management Mode (SMM), a highly privileged execution mode of x86 microprocessors. Our model and proofs have been implemented using Coq, a proof assistant system and have been released as an open source software [3]. We discuss our results in Section 5, some related works in Section 6 and conclude in Section 7.

2 The SpecCert Formalism

In SpecCert, we model the hardware architecture and its features with a set of states $\mathcal{H}$, a set of events $\mathcal{E}$ and a Computing Platform Σ which defines a semantics of events as state-transformers. Hence, the execution of a set of software components by a hardware architecture is a sequence of state-transformations (denoted $h \xrightarrow[\Sigma]{ev} h'$) in this model. In this paper, we consider exclusively Execution Monitoring (EM) enforceable security policies [24,4] that are security policies which can be enforced by monitoring the software execution. As a consequence, we model a security policy with a predicate P on sequences of state-transformations. Finally, we model a HSE mechanism Δ with a set of requirements on states to characterize safe hardware configurations and a set of requirements on state-transformations for trusted software components to preserve the state requirements through software execution. A HSE mechanism is sound when every sequence of state transformations which satisfies these requirements also satisfies the security policy predicate.

[3] Which can can be found at: `https://github.com/lethom/speccert`

2.1 Computing Platforms

We now dive more deeply into the SpecCert formalism and give a formal definition of the Computing Platform. We model a hardware architecture which executes several software components using states, events and a semantics of events as states-transformers.

The state of a hardware architecture models the configuration of its devices at a given time. This configuration may change over time with respect to the hardware specifications and comprises any relevant data such as registers values, inner memory contents, etc. A hardware architecture state update is triggered by some events. We distinguish two classes of events: the software events which are direct and forseeable side-effects of the execution of an instruction and the hardware events which are not. The execution of an instruction can be broken down into a sequence of software events.

For instance, to execute the x86 instruction [4] `mov (%ecx),%eax`, a x86 CPU:

- reads the content of the register `ecx` as an address
- reads the main memory at this address
- writes this content into the register eax
- updates the register `eip` with the address of the next instruction to execute

We model this sequence of actions as four software events which trigger four state updates. Note that if the content of the `ecx` register is not a valid address, the scenario is different. In such a case, the read access to the main memory fails and an interrupt is raised. This second scenario is modeled with another sequence of events which involved a hardware event *i.e.* the interrupt.

The semantics of events as state-transformers is specified using preconditions and postconditions. Preconditions specify the state requirements which are necessary for an event to be observed. Postconditions specify the consequences of an event on the hardware architecture state.

Definition 1 (Computing System). Given $\mathcal{H}$ a set of hardware architecture states and $\mathcal{E}$ a set of events, a Computing Platform Σ is a pair of $(precondition, postcondition)$ where $precondition$ is a predicate on $\mathcal{H} \times \mathcal{E}$ and $postcondition$ is a predicate on $\mathcal{H} \times \mathcal{E} \times \mathcal{H}$. Σ defines a semantics of events as state-transformers such as

$$\frac{precondition(h, ev) \qquad postcondition(h, ev, h')}{h \xrightarrow[\Sigma]{ev} h'}$$

$h \xrightarrow[\Sigma]{ev} h'$ is called a state-transformation of Σ.

2.2 Security Policies

Given $\mathcal{H}$ a set of states of a hardware architecture, $\mathcal{E}$ a set of events, Σ a Computing Platform and $\mathcal{S}$ a set of software components being executed by the hardware architecture, a particular execution of a set of software components is modeled with a sequence of state-transformations we call a run of Σ.

[4] Written in AT&T syntax here.

Definition 2 (Run). A run of the Computing Platform Σ is a sequence of state-transformations of Σ such that for two consecutive transformations, the resulting state of the first is the initial state of the next. We denote $\mathcal{R}(\Sigma)$ the set of runs of the Computing Platform Σ and $init(\rho)$ the initial state of a run ρ.

We consider EM-enforceable security policies [24,4] specified with predicates on runs. A run is said to be secure according to a security policy when it satisfies the predicate specifying this policy.

In this paper, we focus on a class of security policies we call software execution isolation policies. Such a policy prevents a set of untrusted software components to tamper with the execution of another set of so-called trusted software components. We consider that a software component tampers with the execution of another when it is able to make the latter execute an instruction of its choice.

In practice, a subset of states of the hardware architecture is dedicated to each software component. For instance, the x86 CPU has a feature called protection rings where each ring can be seen as an execution mode dedicated to a software component. Hence, the ring 0 is dedicated to the operating system whereas the userland applications are executed when the CPU is in ring 3. In SpecCert, we take advantage of this CPU state sharing to infer which software component is currently executed from a hardware architecture state. For the following definitions, we assume the hardware architecture contains only one CPU.

Definition 3 (Hardware-Software Mapping). A hardware-software mapping $context : \mathcal{H} \rightarrow \mathcal{S}$ is a function which takes a hardware state and returns the software component currently executed.

Dealing with multi-core architectures would require additional efforts and notations. One possible solution could be to define an identifier per core and to use this identifier in addition to the current hardware state to deduce the software component currently executed by the corresponding core. However, this is out of the scope of this article.

We now introduce the concept of *memory location ownership*. A memory location within a hardware architecture is a container which is able to store data used by a software component *e.g.* a general-purpose register of a CPU, a DRAM memory cell, etc. We say that a Computing Platform tracks the memory location ownership if the hardware architecture states maps each memory location with a software component called its *owner*, and the Computing Platform semantics updates this mapping through state-transformations. A software component becomes the new owner of a memory location when it overrides its content during a state-transformation. By extension, we say a software component owns some data when it owns the memory location in which these data are stored.

With this mapping, it becomes possible to determine the owner of an instruction fetched by the CPU in order to be decoded and executed.

Definition 4 (Event-Software Mapping). An event-software mapping $fetched : \mathcal{H} \times \mathcal{E} \rightarrow \mathcal{P}(\mathcal{S})$ is a function which takes an initial hardware state and an

event and returns the set of the fetched instructions owners during this state-transformation.

Hence, $s \in fetched(h, ev)$ means that an instruction owned by s was fetched during a state-transformation triggered by an event ev from a state h. With a hardware-software mapping and an event-software mapping, we give a formal definition of a *software execution tampering*.

Definition 5 (Software Execution Tampering). Given h the initial state of a state-transformation triggered by an event ev, *context* a hardware-software mapping, $fetched$ an event-software mapping and $x, y \in \mathcal{S}$ two software components, the software component y tampers with the execution of another software component x if the CPU fetches an instruction owned by y in a state dedicated to x.

$$software\_tampering(context, fetched, h, ev, x, y) \triangleq \\ context(h) = x \wedge y \in fetched(h, ev)$$

Given $\mathcal{T} \subseteq \mathcal{S}$ a set of trusted software components, the software execution isolation policy prevents the untrusted components from tampering with the execution of the trusted components. Such a policy is enforced during a run if no untrusted component is able to tamper with the execution of a trusted component.

Definition 6 (Software Execution Isolation). Given *context* a hardware-software mapping, $fetched$ an event-software mapping and ρ a run of Σ,

$$software\_execution\_isolation(context, fetched, \rho, \mathcal{T}) \triangleq \\ \forall h \xrightarrow[\Sigma]{ev} h' \in \rho, \forall t \in \mathcal{T}, \forall u \notin \mathcal{T}, \\ \neg software\_tampering(context, fetched, h, ev, t, u)$$

In this definition, t is a trusted software component and u is an untrusted — potentially malicious or hijacked — one.

2.3 Hardware-based Security Enforcement Mechanism

A HSE mechanism is a set of requirements on states to characterize safe hardware configurations and a set of requirements on state-transformations to preserve the state requirements through software execution. The software components which implement a HSE mechanism form the Trusted Computing Base (TCB).

Definition 7 (HSE Mechanism). Given $\mathcal{H}$ a set of states of a hardware architecture, $\mathcal{E}$ a set of events and Σ a Computing Platform, we model a HSE mechanism Δ with a tuple $(inv, behavior, \mathcal{T}, context)$ such as

- inv is a predicate on $\mathcal{H}$ to distinguish between safe hardware configurations and potentially vulnerable ones
- $behavior$ is a predicate on $\mathcal{H} \times \mathcal{E}_{Soft}$ to distinguish between safe software state-transformations and potentially harmful ones

- $\mathcal{T} \subseteq S$ is the set of software components which form the TCB of the HSE mechanism
- *context* is a hardware-software mapping to determine when the TCB is executed

For instance, in x86-based hardware architectures, the SPI Flash contents (the code and configuration of the firmware) is protected as follows:

1. By default, the SPI Flash is locked and its content cannot be overriden until it has been unlocked
2. Some software components can unlock the SPI Flash
3. When they do so, the CPU is forced to start the execution of a special-purpose software component
4. This software component has to lock the SPI Flash before the end of its execution

In this example, the special-purpose software component is the TCB. A safe hardware state (modeled with *inv*) is either a state wherein the special-purpose software component is executed or a state wherein the SPI Flash is locked. This requirement on hardware architecture states is preserved by preventing the special-purpose software component to end its execution before it has locked the SPI Flash (modeled with *behavior*).

For a HSE mechanism to be correctly defined, it must obey a few axioms, together called the HSE Laws. The first law says that the state requirements specified by *inv* are preserved through state-transformations if the software transformations which do not satisfy *behavior* are discarded. The second law says that the *behavior* predicate specifies state-transformations restrictions for the TCB only. The software components which are not part of the TCB are considered untrusted and we make no assumption on their behavior.

Definition 8 (HSE Laws). A HSE mechanism $\Delta = (inv, behavior, \mathcal{T}, context)$ has to satisfy the following properties:

1. *behavior* preserves *inv*: $\forall h \xrightarrow[\Sigma]{ev} h'$,

$$inv(h) \Rightarrow (ev \in \mathcal{E}_{Soft} \Rightarrow behavior(h, ev)) \Rightarrow inv(h')$$

2. *behavior* only restricts the TCB: $\forall x \notin \mathcal{T}, \forall h \in \mathcal{H}, \forall ev \in \mathcal{E}_{Soft}$,

$$context(h) = x \Rightarrow behavior(h, ev)$$

A run complies to a HSE mechanism definition if its initial state satisfies the state requirements and each state-transformation of the run satisfies the state-transformations requirements. The set of the runs which comply with Δ is denoted by $\mathcal{C}(\Delta)$.

Definition 9 (Compliant Runs). Given $\rho \in \mathcal{R}(\Sigma)$,

$$\rho \in \mathcal{C}(\Delta) \triangleq inv(init(\rho)) \wedge \forall h \xrightarrow[\Sigma]{ev} h', ev \in \mathcal{E}_{Soft} \Rightarrow behavior(h, ev)$$

Eventually, we aim to prove that a HSE mechanism is sound —it succeeds to enforce a security policy— under the assumption that software components of the TCB always behave according to the specification given in the HSE mechanism definition.

Definition 10 (Sound HSE Mechanism). A HSE mechanism Δ succeeds in enforcing a security policy P when each compliant run of Δ is secure. In such a case, Δ is said to be sound.

$$sound(\Delta, P) \triangleq \forall \rho \in \mathcal{C}(\Delta), P(\rho)$$

Notation	Description
$\mathcal{S}$	Set of software components
$\mathcal{H}$	Set of states of the hardware architecture
$\mathcal{E}$	Set of states of events of the hardware architecture
Σ	Semantic of events as state-transformers (Computing Platform)
$h \xrightarrow[\Sigma]{ev} h'$	State-transformation according to Σ
$\mathcal{R}(\Sigma)$	Set of sequences of Σ state-transformations (Runs)
P	Predicate on run to model a EM-enforceable security policy
Δ	Requirements on states and state-transformation (HSE mechanism)

Table 1. SpecCert CheatSheet

3 Minx86: a x86 Model

The SpecCert formalism is the foundation of the SpecCert framework. It comprises a set of high-level definitions to specify a HSE mechanism against a hardware architecture model. In its current state, the SpecCert framework contains a model of x86 called Minx86. Minx86 is intended to be a minimal model for single core x86-based machines and we have used publicly available Intel documents [10,11,12] to define it.

3.1 Model Scope

The hardware architecture we are modeling with Minx86 contains a CPU, a cache, a memory controller, a DRAM controller and a VGA controller [5] which both expose some memory to the CPU.

Minx86 is meant to be a proof of concept of the SpecCert formalism and thus is not exhaustive. In its current state of implementation, its scope focuses on the System Management Mode (SMM) feature of x86 microprocessors.

[5] A VGA controller is a hardware device which on we can connect a screen. It exposes some memory to the CPU for communication purposes.

Hardware Specifications We consider the CPU can be either in System Management Mode (SMM) or in an unprivileged mode. The SMM is "a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code" [12]. It is the most privileged execution mode of x86 processors. When a CPU receives a special hardware interrupt called System Management Interrupt (SMI), it halts its current execution and reconfigures itself to a specified state from which it executes the code stored in memory at the address $SMBASE + \texttt{0x8000}$. In practice, the SMBASE value points to the base of a memory region called the SMRAM. Leaving the SMM is done by executing a special purpose instruction called `rsm` (for *resume*).

The CPU relies on a cache to reduce the Input/Output (I/O, that is a read or write access to the memory) latency. We model one level of cache which stores both data and instructions and we consider two cache strategies: uncacheable (UC) and writeback (WB). With the UC cache strategy, the cache is not used and all I/O-s are forwarded to the memory controller, whereas with the WB strategy, the cache is used as much as possible [6]. To determine which cache strategy to use, the CPU relies on several configuration registers and mechanisms. One of them is a pair of registers called the System Management Range Registers (SMRR) which can only be configured when the CPU is in SMM. They are used to tell the CPU where the SMRAM is and which cache strategy to use for I/O targeting the SMRAM when the CPU is in SMM. When it is not in SMM, the CPU always uses the UC strategy for I/O targeting the SMRAM. SMRR have been introduced as a countermeasure of the SMRAM cache poisoning attack [17,22] which allowed an untrusted code to tamper with the copy of the SMRAM stored in the cache. The memory controller [11] receives all the CPU I/O-s which are not handled by the cache and dispatches them to the DRAM controller or to the VGA controller. It exposes a unified view (the memory map) of the system memory to the CPU. The CPU manipulates this memory map with a set of addresses called the physical addresses. The memory controller dedicates a special range of physical addresses to form the SMRAM. The SMRAM is dedicated to store the code intended to be executed when the CPU is in SMM.

Tracking the Memory Ownership The Minx86 definition is parameterized with an hardware-software mapping (see Definition 3). The memory locations of Minx86 Computing Platforms are either cache lines or memory cells exposed by the DRAM controller or the VGA controller. The memory ownership is updated through state-transformations according to three rules:

1. When a cache line gets a copy of a DRAM or VGA cell content, the owner of this cell becomes the new owner of this cache line.
2. When the content of this cache line is written back to a memory cell, the new owner of this memory cell is the owner of this cache line.

[6] These cache strategies are explained in [12], Volume 3A, Chapter 11, Section 11.3 (page 2316 – 2317)

3. When a state-transformation implies the content of a memory location to be overriden with a new value, the software currently executed becomes its new owner.

Given $\mathcal{S}$ a set of software components, the set of states of MINX86 Computing Platform hardware architecture is denoted by $\texttt{Archi}_{\mathcal{S}}$ and the set of MINX86 Computing Platform events is denoted by `Event`. Given *context* a hardware-software mapping, we denote the Computing Platform MINX86 parameterized with *context* [7] such that

$$\text{MINX86}(context) \triangleq (minx86\_pre, minx86\_post(context))$$

3.2 Hardware Architecture State

$\texttt{Archi}_{\mathcal{S}}$ is defined as the Cartesian product of the set of states of the CPU, the CPU's cache, the memory controller and the hardware memories exposed by both the DRAM controller and the VGA controller. Each of these sets is defined in order to model the hardware features we have previously described. We define $\texttt{PhysAddr} \triangleq \{\, \texttt{pa}_i \mid i \leq \texttt{max\_addr} \,\}$ the set of physical addresses the CPU uses to perform I/O. The maximal address offset (denoted by `max_addr` here) is specific to the CPU and may vary in time according to its addressing mode (real mode, long mode, etc.), therefore we left its value as a parameter of our model. An in-depth definition of $\texttt{Archi}_{\mathcal{S}}$ is given in appendix A.1.

We model the projection of the SMRAM in the memory map such that $\texttt{pSmram} \triangleq \{\, \texttt{pa}_i \mid \texttt{smram\_base} \leq i \leq \texttt{smram\_end} \,\}$. The values of `smram_base` and `smram_end` are specified in the memory controller specifications. It is the software responsability to set the SMRR accordingly. We assume $\texttt{smram\_end} - \texttt{smram\_base} > \texttt{0x8000}$. This way, when the SMBASE contains the address of the beginning of the SMRAM, the SMM entry point (that is $SMBASE + \texttt{0x8000}$) is in SMRAM.

The hardware architecture states are implemented in the *SpecCert.x86.Architecture* module (about 1 500 lines of code).

3.3 Events as State-Transformers

The set of events which trigger the state-transformations is denoted by `Event`. As we said in Section 2.1, we distinguish hardware events denoted by $\texttt{Event}_{Hard}$ and software events denoted by $\texttt{Event}_{Soft}$.

Table 2 lists the software events we consider in the MINX86 Computing Platforms. We model the CPU I/O-s with $Read(pa)$ and $Write(pa)$, the configuration of the memory controller with $OpenBitFlip$ and $LockSmramc$, the configuration of the cache strategy with $SetCacheStrat(pa, strat)$, the configuration of the SMRR with $UpdateSmrr(smrr)$ the exit of the SMM with Rsm and the update of the CPU program counter register with $NextInstruction(pa)$.

[7] The related definitions and explanations are given on page 10.

Event	Paramters	Description
$Write$	$pa \in$ `PhysAddr`	A CPU I/O to write at physical address pa
$Read$	$pa \in$ `PhysAddr`	A CPU I/O to read at physical address $pa \in$ `PhysAddr`
$SetCacheStrat$	$pa \in$ `PhysAddr` $strat \in \{$ `UC`, `WB` $\}$	Change the cache strategy for pa to $strat$ (`WB` means write-back and `UC` means uncacheable)
$UpdateSmrr$	$smrr \in$ `Smrr`	Update the SMRR content with the new value $smrr$
Rsm	—	The CPU leaves SMM
$OpenBitFlip$	—	Flip the d_open bit
$LockSmramc$	—	Set the d_lock bit
$NextInstruction$	$pa \in$ `PhysAddr`	The program counter register of the CPU is set to pa

Table 2. List of software events

Event	Description
$Fetch$	A CPU I/O to fetch the instruction stored at the physical address contained in the program counter register
$ReceiveSmi$	A SMI is raised and the CPU handles it

Table 3. List of hardware events

The other causes of state-transformations are modeled using hardware events. Table 3 lists the hardware events we consider in the MINX86 Computing Platforms. $Fetch$ models the I/O to fetch the instruction pointed by the program counter register. $ReceiveSmi$ models a System Management Interrupt being risen and handled by the CPU.

We define $minx86\_fetched$ an event-software mapping for MINX86 Computing Platforms (see Definition 4). The $minx86\_fetched$ function maps a state-transformation to the set of software components which own an instruction fetched during this state-transformation. In the case of MINX86, there is only one event which implies fetching instructions: $Fetch$. Let o be the owner of the instruction pointed by the program counter register in the formula

$$minx86\_fetched(h, ev) \triangleq \begin{cases} \{ o \} & \text{if } ev = Fetch \\ \emptyset & \text{otherwise} \end{cases}$$

We can determine o because MINX86 tracks the memory location ownership.

Given *context* a hardware-software mapping (see Definition 3), the precondition predicate of the Computing Platform MINX86(*context*) is named $minx86\_pre$ and the postcondition predicate is named $minx86\_post(context)$. We give an informal description of the $minx86\_pre$ and $minx86\_post(context)$ for each event. These definitions have been implemented in Coq in the module *SpecCert.x86.Transition*.

We first give the semantics of software events as state-transformers. A software component can always read and write at any physical address. As a consequence, the precondition for $Read(pa)$ and $Write(pa)$ always holds true. The postcondition for $Read(pa)$ and $Write(pa)$ requires the memory ownership to be updated according to the memories and cache state updates. The memory controller enforces a simple access control to protect the SMRAM content in the DRAM memory by forwarding the related I/O to the VGA controller when the CPU is not in SMM. To determine the owner of the memory location which sees its content overriden during a state transformation, the postcondition uses the hardware-software mapping used to define the Computing Platform.

A software component can always update the cache strategy used for an I/O. The postcondition for $SetCacheStrat(pa, strat)$ requires only the cache strategy setting for this physical address pa to change. The precondition for $UpdateSmrr$ requires the CPU to be in SMM. The postcondition requires the SMRR of the CPU to be updated with the correct value, the rest of the hardware architecture state being left unchanged.

A software component can jump to any physical address, hence the postcondition for $NextInstruction(pa)$ always holds true. The postcondition for $NextInstruction(pa)$ requires the program counter register to be updated with pa. The *OpenBitFlip* precondition requires the SMRAMC register to be unlocked. The postcondition requires the `d_open` bit to be updated. The *Lock-Smramc* precondition requires the `d_lock` bit to be unset. The postcondition requires the `d_open` bit to be unset and the d_lock bit to be unset.

We now describe the semantics of hardware events as state-transformers. $Fetch$ models the fetching of an instruction by the CPU. As a consequence, the definition of its precondition and postcondition are the same as $Read(pa)$ with pa being the program register value. *ReceiveSmi* precondition requires the CPU not to be in SMM because SMM is non-reentrant. The postcondition of *ReceiveSmi* requires the program counter to be set with the $smbase + \texttt{0x8000}$ (where $smbase$ is the value of the SMBASE register of the CPU) and the CPU is in SMM.

4 System Management Mode HSE

In [12], Intel states "the main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications". For the SMM processor environment to be isolated, the code executed when the CPU is in SMM needs to implement a HSE mechanism. In this section, we formalize and verify this mechanism against the model we have previously introduced.

4.1 Computing Platform and Security Policy

We consider three software components: the boot sequence code, the SMM code and the OS code. During the boot sequence, only the boot sequence code is

executed and it loads both the OS code and the SMM code into memory. At the end of the boot sequence, the OS kernel is executed. This OS kernel will schedule different applications. Because applications are less privileged than the OS kernel, we will not distinguish them from the kernel code. Thus, in the following, OS code refers to both OS kernel and application codes.

At runtime, both the OS code and the SMM code can be executed. Our objective is to evaluate the security provided by the hardware to isolate SMM code from OS code. Thus, we define

$$\mathcal{S} \triangleq \{\, \texttt{smm}, \texttt{os} \,\}$$

We assume the SMM is dedicated to the SMM code. Let cpu_in_smm : $\texttt{Archi}_{\mathcal{S}} \rightarrow \{\, \texttt{true}, \texttt{false} \,\}$ be the function which returns `true` if the CPU is in SMM and `false` otherwise. We define $smm\_context$ a hardware-software mapping such that

$$smm\_context(h) \triangleq \begin{cases} \texttt{smm} & \text{if } cpu\_in\_smm(h) = \texttt{true} \\ \texttt{os} & \text{otherwise} \end{cases}$$

Let SMMX86 be the Computing Platform such as

$$\text{SMMX86} \triangleq \text{MINX86}(smm\_context)$$

We assume that both the OS code and the SMM code have been loaded in distinct memory regions. In particular, all the SMM code has been loaded in SMRAM. Our objective is to enforce a security policy which prevents the OS code to tamper with the SMM code execution. This way, the SMM (which is the most privileged execution mode of the CPU) cannot be used to perform an escalation privilege. We define $smm\_security$ a predicate to model this security policy such as given $\rho \in$ SMMX86,

$$smm\_security(\rho) \triangleq$$
$$software\_execution\_isolation(smm\_context, minx86\_execute, \rho, \{\, \texttt{smm} \,\})$$

4.2 HSE Definition

We define Δ_{Smm} to model the HSE mechanism applied by the SMM code such that $\Delta_{Smm} = (inv_{Smm}, behavior_{Smm}, \{\, \texttt{smm} \,\}, smm\_context)$ (see Definition 7).

In order to enforce the SMM security policy, we have identified six requirements on states.

- When the CPU executes the SMM code, the program counter register value needs to be an address in SMRAM.
- The SMBASE register was correctly set during the boot sequence to point to the base of the SMRAM.
- The SMRAM contains only SMM code.
- For a physical address in SMRAM, in case of cache hit, the related cache line content must be owned by the SMM code.

– In order to protect the content of the SMRAM inside the DRAM memory, the boot sequence code has locked the SMRAMC controller. This ensures that an OS cannot set the `d_open` bit any longer and only a CPU in SMM can modify the content of the SMRAM.
– The range of memory declared with the SMRR needs to overlap with the SMRAM.

The Appendix A.2 gives the formal definitions of each requirements and of inv_{Smm}. We now define $behavior_{Smm}$. We only define two restrictions. First, we force the SMM code execution to remain confined within the SMRAM. The reason is simple: the OS code can tamper with the memory outside the SMRAM. As a consequence, jumping outside the SMRAM is the best way to fail the security policy. Secondly, we prevent the SMM code to update the SMRR registers as it is the responsability of the boot sequence code to correctly set them.

$$\begin{array}{rl} & smm\_context(h) = \texttt{smm} \\ behavior_{Smm}(h, ev) \triangleq & \Rightarrow ((e = NextInstruction(pa) \Rightarrow pa \in \texttt{pSmram}) \\ & \quad \wedge (e \neq UpdateSmrr(smrr))) \end{array}$$

For Δ_{Smm} to be a HSE mechanism, we need to prove the two HSE Laws (see Definition 8). The first law states the state requirements modeled with inv_{Smm} are preserved through state-transformations if the transformations which do not satisfy $behavior_{Smm}$ are discarded. We prove this by enumeration of $ev \in \texttt{Event}$ and $h \in \texttt{Archi}_{Smm}$, we check that each requirement described previously is preserved by Δ_{Smm}. We use those intermediary results to conclude. The second law states that the $behavior_{Smm}$ predicate specifies state-transformation requirements for the TCB only. In this use case, it means $behavior_{Smm}$ should always hold true when the OS code is executed by the hardware architecture. By definition of $behavior_{Smm}$, $smm\_context(h) = \texttt{smm}$ is an antecedent of the conditional.

Let $smm\_secure\_transformation$ be a predicate which holds true when a state-transformation does not imply the OS code to tamper with the execution of SMM code.

$$\begin{array}{l} smm\_secure\_transformation(h, ev) \triangleq \\ \quad \neg software\_tampering(smm\_context, minx86\_execute, h, ev, \texttt{os}, \texttt{smm}) \end{array}$$

We prove that this predicate holds true for a state-transformation with respect to the HSE mechanism. With this result, we can prove the HSE mechanism is sound (see Definition 10).

Lemma 1 (Invariants Enforce Security). $\forall h \xrightarrow[\textsc{Smmx86}(ctx)]{ev} h'$,

$$\begin{array}{l} inv_{Smm}(h) \\ \quad \Rightarrow (ev \in \texttt{Event}_{Soft} \Rightarrow behavior_{Smm}(h, ev) \\ \quad\quad \Rightarrow smm\_secure\_transformation(h, ev) \end{array}$$

Proof. By enumeration of $ev \in \texttt{Event}$ and $h \in \texttt{Archi}_{\mathcal{S}}$.

Theorem 1 (Δ_{Smm} is Sound).

$$sound(\Delta_{Smm}, smm\_security)$$

Proof. The "Invariants Enforce Security" lemma applies for one transition and the first HSE law allows to reason by induction on runs.

5 Discussion

Our effort has been originally motivated by the disclosure of several vulnerabilities targeting multiple x86 HSE mechanisms for the past few years [17,22,23,6,14]. These attacks do not benefit from a software implementation error but rather from a flaw in the hardware specifications themselves. The result of our work is a three-steps methodology for formally specifying and verifying HSE mechanisms against a hardware architecture model. We believe each aspect is important.

First, the hardware architecture model can be used as a formal specification. The main benefit of a formal specification is to avoid any ambiguity such as the one we have found in [11]. One can read at Section 3.8.3.8, page 102 that "the OPEN bit must be reset before the LOCK bit is set". At the same page, in the description of the LOCK bit, one can also read that "when [LOCK] is set to 1 then [OPEN] is reset to 0". We had modeled the second statement as the behavior of the memory controller is not specified if the first statement is true [8] MINX86 as a formal specification does not suffer from the same flaw. However, the scalability of our approach remains to be proven. MINX86 is far from being complete, as it focuses on SMM-related mechanisms. It is hard to forsee the amount of work required to write both a more realistic hardware model and the related security proofs.

Secondly, a formal specification of a HSE mechanism will help software developers when the time comes to implement it. For instance, the Chapter 34, Volume 3C of [12] about SMM is about 30 pages long, it gives many details on how the SMM actually works, yet no section is actually dedicated to security. On the contrary, our HSE mechanism definition gathers six requirements on hardware configurations and two requirements on software executions to enforce a well-defined security property. Even if the proofs only apply to an abstract model, we believe it is a valuable improvement.

Lastly, the verification process of a HSE mechanism specification against a hardware architecture model may help to highlight hidden flaws in the hardware specifications assumptions. We take the example of the SMRAM cache poisonning attack [17,22], which has motivated the introduction of the SMRR. If an attacker can set the proper cache strategy (WB) for the SMRAM physical addresses, then the code inside the SMRAM is loaded into the cache as soon as the CPU in SMM is executing it. From this point forward —because the access control is enforced at the memory controller level— nothing prevents the attacker

[8] If we had to actually implement the HSE mechanism, we would have to assume the first was the correct one.

to tamper with it. The next time the CPU enters in SMM, it executes the code stored in the cache. With a SMRR-less version of MINX86, we were not able to conclude our HSE mechanism was sound: such a scenario draws attention of the SpecCert user who is forced to investigate.

From our point of view, the clear separation between the hardware model, the security properties and the HSE mechanisms to enforce those properties are the main advantage of our approach, as two different use cases can be studied against the same hardware model.

6 Related Works

Several formal models of x86 architectures have been defined. For instance, Greg Morrisett *et al.* have developed RockSalt [20], a sandboxing policy checker, upon such a model. Peter Sewell *et al.* have proposed a model for x86 multiprocessors [25] which aims at replacing informal Intel and AMD specifications. Andrew Kennedy *et al.* have developed an assembler in Coq [15] which allows a developer to verify the correctness of a specification for an assembly code. These three projects have modeled (a subset of) the x86 instruction set against an idealized hardware. Our approach is different: we model the instructions' side effects on a hardware architecture model as close as possible to its specifications.

Our work is inspired by the efforts by Gilles Barthe *et al.* to formally verify an idealized model of virtualization [1,2,3]. In this work, the authors have developed a model of a hypervisor and have verified that the latter correctly enforces several security properties among which the guest OSes isolation. From the SpecCert perspective, a hypervisor relies on HSE mechanisms which could be specified and verified using SpecCert and a more complete version of the MINX86 model.

To the best of our knowledge, the closest related research project is the work of David Lie *et al.* They have used a model checker (Murφ) to model and verify the eXecute Only Memory (XOM) architecture [16]. The XOM architecture allows an application to run in a secure compartment wherein its data are protected against other applications and even a malicious operating system. The main difference with our approach is that the XOM security properties are enforced as-is by a secure microprocessor without the need for a software component to configure anything. On the contrary, we intend to specify ways to use sets of hardware functions to enforce security policies.

From our point of view, the main limitation of the research previously described, including SpecCert, is the gap between the model and the concrete machine. The recent efforts around the Proof Carrying Hardware (PCH) [18,19,8], inspired by the research about Proof Carrying Code (PCC), is promising. The main idea behind PCH is to derive a model from a hardware device implementation written in a Hardware Description Language (HDL). One of our objective is to investigate the possibility to adapt the SpecCert formalism to the PCH models.

7 Conclusion

In this paper, we have focused on a class of security enforcement mechanism we called Hardware-based Security Enforcement (HSE). The contribution of this article is threefold. First, we have proposed a formalism to specify and verify HSE mechanisms against hardware architecutre models. Then, we have defined a minimalist x86 model called MINX86. Finally, we have specified and verified the HSE mechanism dedicated to enforce the SMM code execution isolation against this model. Our model and proofs have been implemented in Coq [9]. The project is about 4 500 Lines of Code (LoC) including 190 definitions and 150 proofs (theorems and lemmas).

For now, our proofs are built against an abstract model of the hardware architecture. One of the future work we aim to address is improving the scope of MINX86 in order to provide to potential SpecCert users a more complete model to use for verifying and specifying their x86-based HSE mechanisms. Ultimately, we aim to extend these proofs to a physical hardware platform. Therefore, the equivalence between the model and the implementation has to be established. In this perspective, the Proof Carrying Hardware framework [7,18,19,8] is particulary interesting and we intend to investigate in this direction.

References

1. Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Formally verifying isolation and availability in an idealized model of virtualization. In: FM 2011: Formal Methods, pp. 231–245. Springer (2011)
2. Barthe, G., Betarte, G., Campo, J.D., Luna, C.: Cache-leakage resilient os isolation in an idealized model of virtualization. In: Computer Security Foundations Symposium (CSF), 2012 IEEE 25th. pp. 186–197. IEEE (2012)
3. Barthe, G., Betarte, G., Campo, J.D., Luna, C., Pichardie, D.: System-level non-interference for constant-time cryptography. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1267–1279. ACM (2014)
4. Basin, D., Jugé, V., Klaedtke, F., Zălinescu, E.: Enforceable security policies revisited. ACM Transactions on Information and System Security (TISSEC) 16(1), 3 (2013)
5. Corey Kallenberg, Sam Cornwell, Xeno Kovah, John Butterworth: Setup For Failure: Defeating Secure Boot
6. Domas, C.: The Memory Sinkhole. In: BlackHat USA (july 2015)
7. Drzevitzky, S.: Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration. In: Field Programmable Logic and Applications (FPL), 2010 International Conference on. pp. 255–258. IEEE (2010)
8. Guo, X., Dutta, R.G., Mishra, P., Jin, Y.: Scalable SoC Trust Verification using Integrated Theorem Proving and Model Checking. In: IEEE Symposium on Hardware Oriented Security and Trust. pp. 124–129 (2016)
9. Intel: CHIPSEC: Platform Security Assessment Framework. `http://github.com/chipsec/chipsec`

[9] Our implementation is available here: `https://github.com/lethom/speccert`

10. Intel: Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family
11. Intel: Intel 5100 Memory Controller Hub Chipset
12. Intel: Intel 64 and IA32 Architectures Software Developer Manual
13. Intel: Intel Trusted Execution Technology (Intel TXT) (07 2015)
14. Kallenberg, C., Wojtczuk, R.: Speed racer: Exploiting an intel flash protection race condition
15. Kennedy, A., Benton, N., Jensen, J.B., Dagand, P.E.: Coq: the world's best macro assembler? In: Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming. pp. 13–24. ACM (2013)
16. Lie, D., Mitchell, J., Thekkath, C., Horowitz, M., et al.: Specifying and verifying hardware for tamper-resistant software. In: Security and Privacy, 2003. Proceedings. 2003 Symposium on. pp. 166–177. IEEE (2003)
17. Loic Duflot, Olivier Levillain, Benjamin Morin, Olivier Grumelard: Getting into the SMRAM: SMM reloaded CanSecWest
18. Love, E., Jin, Y., Makris, Y.: Proof-carrying hardware intellectual property: A pathway to trusted module acquisition. Information Forensics and Security, IEEE Transactions on 7(1), 25–40 (2012)
19. Makris, Y.: Trusted module acquisition through proof-carrying hardware intellectual property. Tech. rep. (2015)
20. Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.B., Gan, E.: Rocksalt: better, faster, stronger sfi for the x86. In: ACM SIGPLAN Notices. vol. 47, pp. 395–404. ACM (2012)
21. Rafal Wojtczuk, Joanna Rutkowska: Attacking intel TXT via SINIT code execution hijacking
22. Rafal Wojtczuk, Joanna Rutkowska: Attacking SMM memory via intel CPU cache poisoning
23. Rutkowska, J., Wojtczuk, R.: Preventing and detecting xen hypervisor subversions. Blackhat Briefings USA (2008)
24. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) 3(1), 30–50 (2000)
25. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. Communications of the ACM 53(7), 89–97 (2010)
26. Yuriy Bulygin, John Loucaides, Andrew Furtak, Oleksandr Bazhaniuk, Alexander Matrosov: Summary of Attacks Against BIOS and Secure Boot, def Con 22

A Appendix

A.1 Minx86 Set of States

To describe the hardware architecture state of Minx86 Computing Platforms, we use record types and maps with the notations listed in Table 4. Given $\mathcal{S}$ a set of software components, the set of states of the hardware architecture is denoted by $\texttt{Archi}_{\mathcal{S}}$.

$$\texttt{Archi}_{\mathcal{S}} \triangleq \begin{array}{ll} \{proc & : \texttt{Proc} \\ ; mc & : \texttt{MC} \\ ; mem & : \texttt{Mem}_{\mathcal{S}} \\ ; cache & : \texttt{Cache}_{\mathcal{S}} \,\} \end{array}$$

Notation	Description
$A \triangleq \{\, field_1 : T_1 \; ; \; field_2 : T_2 \,\}$	Set of records definition
$field_1(a)$	Field selection
$M \triangleq [A \to B]$	Set of maps definition
$m[a]$	Map application

Table 4. List of notations

$\texttt{Proc}$ denotes the CPU set of states, $\texttt{MC}$ the memory controller set of states, $\texttt{Mem}_{\mathcal{S}}$ the physical memories set of states and $\texttt{Cache}_{\mathcal{S}}$ the cache set of states.

We define $\texttt{CacheStrat} \triangleq \{\, \texttt{UC}, \texttt{WB} \,\}$ the set of the modeled cache strategies. The set of states of the SMRRs is denoted $\texttt{Smrr}$.

$$\texttt{Smrr} \triangleq \begin{array}{ll} \{\, range & : \mathcal{P}(\texttt{PhysAddr}) \\ ;\, smram\_strat & : \texttt{CacheStrat} \}\end{array}$$

The set of physical addresses *range* tells the CPU the location of the SMRAM and *smram_strat* tells which cache strategy has to be used when the CPU is in SMM.

$$\texttt{Proc} \triangleq \begin{array}{ll} \{\, in\_smm & : \{\, \texttt{true}, \texttt{false} \,\} \\ ;\, pc & : \texttt{PhysAddr} \\ ;\, smbase & : \texttt{PhysAddr} \\ ;\, smrr & : \texttt{Smrr} \\ ;\, strat & : [\texttt{PhysAddr} \to \texttt{CacheStrat}] \,\}\end{array}$$

The boolean *in_smm* is set to $\texttt{true}$ when the CPU is in SMM and to $\texttt{false}$ when it leaves it. The physical address *pc* models the program counter, a register used to store the address of the next instruction to be fetched and executed. The physical address *smbase* models the register of the same name. The map *strat* abstracts away the numerous mechanisms of the x86 microprocessors to determine which cache strategy to use for a given I/O.

The set of states of the MINX86 Computing Platforms memory controller is denoted by $\texttt{MC}$.

$$\texttt{MC} \triangleq \begin{array}{ll} \{\, d\_open & : \{\, \texttt{true}, \texttt{false} \,\} \\ ;\, d\_lock & : \{\, \texttt{true}, \texttt{false} \,\} \,\}\end{array}$$

The two booleans *d_open* and *d_lock* model two bits of a configuration register named $\texttt{smramc}$. They are used to determine how the memory controller dispatches the I/O which targets a physical address of the SMRAM.

For a memory controller state $mc \in \texttt{MC}$ to be consistent with respect to the hardware specifications, it has to to verify that $d\_lock(mc) = \texttt{true} \Rightarrow d\_open(mc) = \texttt{false}$.

The memory controller translates physical addresses into hardware addresses and forwards the I/O accordingly. We model this translation with the function

$$phys\_to\_hard : \texttt{MC} \times \{\, \texttt{true}, \texttt{false} \,\} \times \texttt{PhysAddr} \rightarrow \texttt{HardAddr}$$

which maps a state of the memory controller, a boolean which tells if the CPU is in SMM or not and a physical address to a hardware address. It is important to keep in mind that the same physical address can be translated into two different hardware addresses for two memory controller states m and m', hence it is possible to have

$$phys\_to\_hard(m, b, pa) \neq phy\_to\_hard(m', b, pa)$$

The physical memories state (exposed by the DRAM controller and the VGA controller) is modeled with a mapping between the hardware addresses and the software component which owns the related memory location. Given $\mathcal{S}$ the set of software components, we denote $\texttt{Mem}_{\mathcal{S}}$ the set of states of the physical memories.

$$\texttt{Mem}_{\mathcal{S}} \triangleq [\texttt{HardAddr} \rightarrow \mathcal{S}]$$

We assume `Index` is the set of cache indexes and $index : \texttt{PhysAddr} \rightarrow \texttt{Index}$ the function used by the CPU to determine which index to use for a given physical address. The cache is divided into several cache lines which contain the cached memory content and several additional information required by the cache strategy algorithm. The set of states of the cache line is denoted by $\texttt{CacheLine}_{\mathcal{S}}$. In addition to modeling the hardware specifications, the definition of $\texttt{CacheLine}_{\mathcal{S}}$ attaches a software owner to a cache line. We do not give more details about the cache line model because the cache behavior is out of the scope of this article.

$$\texttt{Cache}_{\mathcal{S}} \triangleq [\texttt{Index} \rightarrow \texttt{CacheLine}_{\mathcal{S}}]$$

In addition to the state definitions, we have implemented several helper functions and predicates. For instance,

$address\_location\_owner : \texttt{Archi}_{\mathcal{S}} \rightarrow \texttt{PhysAddr} \rightarrow \mathcal{S}$
: Given a hardware architecture state and a physical address, returns the memory content software owner

$cache\_hit : \texttt{Archi}_{\mathcal{S}} \rightarrow \texttt{PhysAddr} \rightarrow \texttt{Prop}$
: Given a hardware architecture state and a physical address, holds true if the memory content is in the cache

$cache\_line\_owner : \texttt{Archi}_{\mathcal{S}} \rightarrow \texttt{Index} \rightarrow \mathcal{S}$
: Given a hardware architecture state and a cache line index, returns the owner of this cache line

$resolve\_cache\_strategy : \texttt{Archi}_{\mathcal{S}} \rightarrow \texttt{PhysAddr} \rightarrow \texttt{CacheStrat}$
: Given a hardware architecture state and a physical address, returns the cache strategy used by the CPU for this address

$translate\_physical\_address : \texttt{Archi}_{\mathcal{S}} \rightarrow \texttt{PhysAddr} \rightarrow \texttt{HardAddr}$
: Given a hardware architecture state and a physical address, returns the result of the memory controller address translation

A.2 SMM Code Isolation: Charactezing a Safe State

In order to enforce the SMM security policy, we have identified six requirements on states.

- When the CPU executes the SMM code, the program counter register value needs to be an address in SMRAM.

$$smram\_pc(h) \triangleq smm\_context(h) = \texttt{smm} \Rightarrow pc(proc(h)) \in \texttt{pSmram}$$

- The SMBASE register has been correctly set during the boot sequence.

$$valid\_smbase(h) \triangleq smbase(proc(h)) = \texttt{pa}_{\texttt{smram\_base}}$$

- The SMRAM must contain only SMM code.

$$smram\_code(s) \triangleq \forall ha \in \texttt{hSmram}, mem(h)[ha] = \texttt{smm}$$

- For a physical address in SMRAM, in case of cache hit, the related cache line content must be owned by the SMM code.

$$cache\_clean(h) \triangleq \begin{array}{c} cache\_hit(h, pa) \\ \Rightarrow cache\_line\_owner(h, index(pa)) \\ = \texttt{smm} \end{array}$$

- In order to protect the content of the SMRAM inside the DRAM memory, the boot sequence code has to lock the SMRAMC controller. This ensures that an OS cannot set the `d_open` bit any longer and only a CPU in SMM can modify the content of the SMRAM.

$$locked\_smramc(h) \triangleq d\_lock(mc(h)) = \texttt{true}.$$

- The range of memory declared with the SMRR needs to overlap with the SMRAM.

$$valid\_smrr(h) \triangleq \texttt{pSmram} \subseteq range(smrr(proc(h)))$$

$$inv_{Smm}(h) \triangleq \begin{array}{l} smram\_pc(h) \\ \wedge\ valid\_smbase(h) \\ \wedge\ smram\_code(h) \\ \wedge\ cache\_clean(h) \\ \wedge\ locked\_smramc(h) \\ \wedge\ valid\_smrr(h) \end{array}$$