# SpecCert: Towards Verified Hardware-based Security Enforcement

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

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

**Abstract.** Over time, hardware designs have constantly grown in complexity and modern platforms involve dozens of interconnected hardware components. During the last decade, several vulnerability disclosures have proven that hardware designs are not immune to misconceptions. In this article, we give a formal definition of *Hardware-based Security Enforcement* (HSE) mechanisms, a class of security enforcement mechanisms where 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 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 enforcement mechanisms [23] we call Hardware-Based Security Enforcement (HSE) where 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 dozens of interconnected devices interacting 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 [12] allowed an attacker to perform a privilege escalation [20]

– The legitimate use of a hardware mechanism can 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 [16,21]

Secondly, hardware architectures have grown in complexity and, as a consequence, HSE mechanisms too. There are many example of security vulnerabilities which are the consequence of an incorret HSE mechanism implementation [5,25,8].

In this paper, we introduce SpecCert, a framework for specifying and verifying HSE mechanisms against hardware architecture models. We have implemented SpecCert using Coq, a proof assistant system. The SpecCert formalism relies on a three-steps methodology. First, we model the hardware architecture specifications. Then, we specify the software requirements of a HSE mechanism by formally defining them in our hardware model. Finally, we prove that the HSE mechanism is sound under the assumption that the software components complies to the specified software requirements. This implies the hardware mechanisms involved in the HSE mechanism indeed provide the security properties they promise. We believe this approach to be beneficial to both hardware designers and software developers. Hardware designers can verify their hardware mechanism assumptions and software developers can get a formalized specification on the use of this mechanism.

*Paper Organization* Our contribution is threefold.

– 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 targetting 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.

We discuss our results in Section 5 and conclude in Section 7.

## 2 The SpecCert Formalism

We have implemented our formalism in SpecCert, our framework written with Coq. In SpecCert, we model the hardware architecture and its behavior with a set of states $\mathcal{H}$, a set of events $\mathcal{E}$ and a Computing Platform $\Sigma$ which defines a semantics of events as state-transformers. Hence, the execution of a set of software components by a hardware platform is a sequence of state-transformations (denoted $h \xrightarrow[\Sigma]{ev} h'$) in this model. We consider Execution Monitoring (EM) enforceable security policies [23,4] 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 $\Delta$ with a set of requirements on states to characterize

safe hardware configurations and a set of requirements on sequences of state-transformations 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.

| 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 |
| $\Sigma$ | Semantic of events as state-transformers (Computing Platform) |
| $h \xrightarrow[\Sigma]{ev} h'$ | State-transformation according to $\Sigma$ |
| $\mathcal{R}(\Sigma)$ | Set of sequences of $\Sigma$ state-transformations (Runs) |
| $P$ | Predicate on run to model a EM-enforceable security policy |
| $\Delta$ | Requirements on states and state-transformation (HSE mechanism) |

**Table 1.** SpecCert Formalism CheatSheet

### 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 an event. We distinguish two classes of events: the software events which are direct and forseeable consequences 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 [3] `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. Indeed, 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

[3] Written in AT&T syntax here.

events. An interrupt is a good example of a hardware event. Another example is a device making a Direct Memory Access (DMA) request.

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 $\Sigma$ is a pair of $(precondition, postcondition)$ where precondition is a predicate on $\mathcal{H} \times \mathcal{E}$ and postcondition is a predicate on $\mathcal{H} \times \mathcal{E} \times \mathcal{H}$. $\Sigma$ 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 $\Sigma$.*

### 2.2 Security Policies

Given $\mathcal{H}$ a set of states of a hardware architecture, $\mathcal{E}$ a set of events, $\Sigma$ 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 within a hardware architecture is modeled with a sequence of state-transformations. We call such a sequence a run of $\Sigma$.

**Definition 2 (Run).** *A run of the Computing Platform $\Sigma$ is a sequence of state-transformations of $\Sigma$ such as 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 $\Sigma$ and $init(\rho)$ the initial state of a run $\rho$.*

In our formalism, we consider EM-enforceable security policies [23,4] 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 especially focus on a class of security policies we call software execution isolation policies. Such 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. We now demonstrate how we can express a software execution isolation policy in our formalism. For the following definitions, we assume the hardware architecture contains only one CPU.

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 the 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 states sharing to infer which software component is currently executed from a hardware architecture state.

**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.*

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.* the general-purpose registers of a CPU, the DRAM memory cells, 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 $s$ has been fetched during a state-transormation 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 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, a software component* $y$ *tampers with the execution of another software component* $x$ *if the CPU executes 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 and $\mathcal{A} \subseteq S$ a set of untrusted software components, the software execution isolation policy which prevents the untrusted components from tampering with the execution of the trusted components 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 $\rho$ a run of $\Sigma$,*

$$\begin{aligned} &software\_execution\_isolation(context, fetched, \rho, \mathcal{T}, \mathcal{A}) \triangleq \\ &\quad \forall h \xrightarrow[\Sigma]{ev} h' \in \rho, \forall t \in \mathcal{T}, \forall a \in \mathcal{A}, \\ &\qquad \neg software\_tampering(context, fetched, h, ev, t, a) \end{aligned}$$

### 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 $\Sigma$ a Computing Platform, we model a HSE mechanism $\Delta$ with a tuple $(inv, behavior, \mathcal{T}, context)$ such as*

- *$inv$ is a predicate on $\mathcal{H}$ to distinguish betweem 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, the SPI Flash content (the firmware code and configuration) 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, the CPU is forced to start the execution of a special-purpose software component
4. This software component has to locked 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 (this is 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 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:* $\forall h, h' \in \mathcal{H}, \forall ev \in \mathcal{E}$,

$$\begin{array}{l} inv(h) \\ \quad \Rightarrow h \xrightarrow[\Sigma]{ev} h' \\ \quad\quad \Rightarrow (ev \in \mathcal{E}_{Soft} \Rightarrow behavior(h, ev)) \\ \quad\quad\quad \Rightarrow inv(h') \end{array}$$

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 $\Delta$ is denoted by $\mathcal{C}(\Delta)$.

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

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

By definition of a run and according to the first HSE law, we prove that each state of a compliant run satisfies the state requirements of the HSE mechanism.

The purpose of SpecCert is 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* $\Delta$ *succeeds in enforcing a security policy* $P$ *when each compliant run of* $\Delta$ *is secure. In such case,* $\Delta$ *is said to be sound.*

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

## 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 cole x86-based machines and we used publicly available Intel DataSheets [9,10] and Specifications [11] to define it.

### 3.1 Model Scope

The hardware architecture we are modeling with MINX86 is summarized in the Figure 1. It contains a CPU, a cache, a memory controller, a DRAM controller and a VGA controller [4] which exposes some memory to the CPU.

MINX86 is far from being an exhaustive model of the x86 specifications. The x86 architecture is far too complex for that (the Intel Software Developer Manual alone is almost 3 500 pages of informal specifications). However, x86 is one of the most heavily studied hardware architecture with many research papers with a security focus which have been valuable inputs for our own reflections. MINX86 is meant to be a proof of concept to show that the SpecCert formalism can be applied to realistic use cases. In its current state of implementation, its scope focuses on the System Management Mode (SMM) feature of x86 microprocessors.

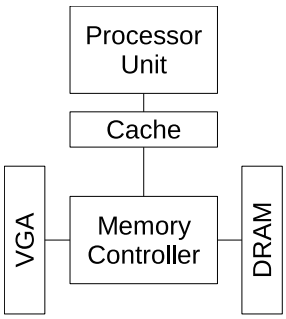


**Fig. 1.** MINX86 Architecture

*Hardware Specifications* The CPU model is based on two documents published by Intel: the 4th Generation Intel Processor Families [9] and the Intel 64 and IA-32 Architectures Software Developer's Manual [11]. We consider the CPU can be either in the System Management Mode (SMM) or in an unprivileged mode when it executes software. The SMM is introduced in the Chapter 34 of [11] as follows:

> SMM is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code.

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 $SMBASE + 0x8000$. In practice, the SMBASE value points to the base of a memory region called the SMRAM. Leaving the SMM is done by executing a

[4] A VGA controller is a hardware device which on we can connect a screen.

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 are forwarded to the memory controller, whereas with the WB strategy, the cache is used as much as possible. UC and WB cache strategies are described along with the other cache strategies of x86 processors in Chapter 11 (Volume 3A) of [11]. To determine which cache strategy has to be used, 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 [16,21] which allowed an untrusted code to tamper with the copy of the SMRAM stored in the cache.

The memory controller [10] receives and dispatches all the CPU I/O-s which are not handled by the cache 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 defines 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.

*Computing Platforms as Functions* In addition to the hardware specifications, some security properties require additional metadata to be defined. For instance, confidentiality properties are expressed by attaching and propagating labels to secrets. Therefore, the definition of the hardware architecture states limits the set of security properties which can be studied with the related Computing Platform.

We have not defined Minx86 as a Computing Platform but as a function which returns a Computing Platform. More precisely, Minx86 allows us to define a x86 Computing Platform which tracks the memory ownership for any set of software component according to a hardware-software mapping (see Definition 3) given in argument to Minx86.

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 the cache line.
2. When the content of the cache line is pushed back to a memory cell, the new owner of this memory cell is the owner of the cache line.

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

**Definition 11 (MINX86).** *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`*. We define* MINX86 *to be a function which takes a hardware-software mapping and returns the Computing Platform which tracks the memory location ownership according to this mapping. Given context a hardware-software mapping,*

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

In the current state of SpecCert implementation, expressing a security policy which requires metadata in addition to software ownership is not yet possible and is left as future work.

### 3.2 Hardware Architecture State

| 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 2.** List of notations

To describe the hardware architecture state of MINX86 Computing Platforms, we use record types and maps with the notations listed in Table 2. Given $\mathcal{S}$ a set of software components, the set of states of the hardware architecture is denoted by $\texttt{Archi}_{\mathcal{S}}$.

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

`Proc` denotes the CPU set of states, `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 `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. As we said, the set of states of the CPU is denoted `Proc`.

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

The boolean *in_smm* is a boolean set to `true` when the CPU is in SMM and to `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 `MC`.

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

The two booleans *d_open* and *d_lock* model two bits of a configuration register named `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 $d\_lock(mc) = \texttt{true} \Rightarrow d\_open(mc) = \texttt{false}$.

We then define several address spaces:

- $\texttt{PhysAddr} \triangleq \{\, \texttt{pa}_i \mid i \leq \texttt{max\_addr} \,\}$ the set of physical addresses the CPU uses to perform I/O.
- $\texttt{DRAMAddr} \triangleq \{\, \texttt{dram}_i \mid i \leq \texttt{max\_addr} \,\}$ the set of addresses of the memory cells exposed by the DRAM controller
- $\texttt{VGAAddr} \triangleq \{\, \texttt{vga}_i \mid i \leq \texttt{max\_addr} \,\}$ the set of addresses of the memory cells exposed by the VGA controller
- $\texttt{HardAddr} \triangleq \texttt{DRAMAddr} \cup \texttt{VGAAddr}$

The maximal address offset (denoted by `max_addr` here) is specific to the CPU (for the physical addresses), the DRAM controller and the VGA controller. By convenience, we give the same values to each of them in our model.

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

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

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

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

We model the SMRAM with two ranges of addresses:

- $\texttt{hSmram} \triangleq \{\, \texttt{dram}_i \mid \texttt{smram\_base} \leq i \leq \texttt{smram\_end} \,\}$ the SMRAM memory range within the DRAM memory
- $\texttt{pSmram} \triangleq \{\, \texttt{pa}_i \mid \texttt{smram\_base} \leq i \leq \texttt{smram\_end} \,\}$ the projection of the SMRAM in the memory map

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 $\texttt{smram\_end} - \texttt{smram\_base} > 0x8000$. This way, when the SMBASE contains the address of the beginning of the SMRAM, the SMM entry point (that is $SMBASE + 0x8000$) is in SMRAM.

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.

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

We assume `Index` is the set of cache indexes and $index : \texttt{PhysAddr} \to \texttt{Index}$ 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.

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

The hardware architecture states are implemented in the *SpecCert.x86.Architecture* module (about 1 500 lines of code). In addition to the state definitions, we have implemented several helper functions and predicates. For instance,

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

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

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

$resolve\_cache\_strategy : \texttt{Archi}_{\mathcal{S}} \to \texttt{PhysAddr} \to \texttt{CacheStrat}$
: 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

### 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 $\texttt{Event}_{Soft}$.

$$\texttt{Event} \triangleq \texttt{Event}_{Soft} \cup \texttt{Event}_{Hard}$$

| Event | Description |
|---|---|
| $Write(pa)$ | A CPU I/O to write at physical address $pa$ |
| $Read(pa)$ | A CPU I/O to read at physical address $pa$ |
| $SetCacheStrat(pa, strat)$ | Change the cache strategy for $pa$ to $strat$ |
| $UpdateSmrr(smrr)$ | Update the SMRR content with a new value |
| $Rsm$ | The CPU leaves SMM |
| $OpenBitFlip$ | Switch the $d\_open$ bit from 0 to 1 or 1 to 0 |
| $LockSmramc$ | The $d\_lock$ bit is set |
| $NextInstruction(pa)$ | The program counter register of the CPU is set to $pa$ |

**Table 3.** List of software events

We model the side effects of the x86 instructions using software events: Table 3 lists the software events we consider in the MINX86 Computing Platforms with a short and informal description of their semantics. 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 $NextIntsruction(pa)$.

| Event | Description |
|---|---|
| $Fetch$ | I/O to fetch an instruction $pa$ |
| $ReceiveSmi$ | A SMI is raised and the CPU handles it |

**Table 4.** List of hardware events

The other causes of state-transformations are modeled using hardware events. Table 4 lists the hardware events we consider in the MINX86 Computing Platforms with a short and informal description of their semantics. $Fetch$ 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: $Fetch$. During a $Fetch$ state-transformation, the CPU fetches the memory at the physical address stored in its program counter register in order to execute it. To define $minx86\_fetched$, we use the function $content\_owner$ we have previously introduced. Let $o$ be $content\_owner(pc(proc(h))$ in the formula

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

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 $minx86\_post(context)$ (see Definition 11). 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. 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 $smrr$ field 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 $LockSmramc$ 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. Note that [10] is ambiguous thereupon. One can read, at Section 3.8.3.8, page 102, that:

> Also, the OPEN bit must be reset before the LOCK bit is set (...)

At the same page, in the description of the LOCK bit:

> When [LOCK] is set to 1 then [OPEN] is reset to 0 (...)

Considering that the behavior of the memory controller is not specified if the first statement is true, we choose to model the second. If we had to actually implement the HSE mechanism, we would have to assume the second. In any case, this situation is yet another argument in favor of formal specifications.

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 $Read(pa)$ for $pa = pc(proc(h))$ ($h$ being the initial state of the state-transformation). $ReceiveSmi$ precondition requires the CPU not to be in SMM, as explained in [11]:

> SMM is non-reentrant; that is, the SMI is disabled while the processor is in SMM.

The postcondition of $ReceiveSmi$ requires the program counter to be set with the $smbase + 0x8000$ value and the $in\_smm$ boolean to be set to `true`.

## 4 System Management Mode HSE

In [11], 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 in memory. At the end of the boot sequence, the OS kernel is executed. This OS kernel will be used to schedule different applications. Even though 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 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. Therefore, we define $smm\_context$ a hardware-software mapping such as

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

Let SMMX86 be the Computing Platform such as

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

We assume 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. We define $smm\_security$ a predicate to model this security policy such as such as given $\rho \in \text{SMMX86}$,

$$\begin{aligned} &smm\_security(\rho) \triangleq \\ &\quad software\_isolation(smm\_context, minx86\_execute, \rho, \{\,\texttt{smm}\,\}, \{\,\texttt{os}\,\}) \end{aligned}$$

### 4.2 HSE Definition

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

*State Invariant* 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.

$$smram\_pc(h) \triangleq \begin{array}{l} smm\_context(h) = \texttt{smm} \\ \quad \Rightarrow pc(proc(h)) \in \texttt{pSmram} \end{array}$$

- 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.

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

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

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

– 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.

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

The Chapter 34, Volume 3C of [11] about SMM is about 30 pages long. It gives many detail on how the SMM actually works, but no section is actually dedicated to security. We had to gather information in many locations and we believe this is another argument in favor of SpecCert.

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

*Software Behavior* We now define $behavior_{Smm}$. We only define two restrictions. First, we force the SMM code execution to remain confined to the SMRAM. The reason is simple: the Adversary 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. It is the responsability of the boot sequence code to correctly set the SMRR.

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

*$\Delta_{Smm}$ is a HSE Mechanism* For $\Delta_{Smm}$ to be a HSE mechanism, we need to prove the two HSE Laws (see Definition 8). The first law states the state requirements model with $inv_{Smm}$ are preserved through state-transformations if the transformations which do not satisfy $behavior_{Smm}$ are discarded.

**Lemma 1 (Invariant Are Preserved).** $\forall h, h' \in \mathit{Archi}_{\mathcal{S}}, \forall ev \in \mathit{Event}$,

$$\begin{array}{l} inv_{Smm}(h) \\ \quad \Rightarrow h \xrightarrow[\textsc{Smmx86}(ctx)]{ev} h' \\ \qquad \Rightarrow (ev \in \mathcal{E}_{Soft} \Rightarrow behavior_{Smm}(h, ev) \\ \qquad\quad \Rightarrow inv_{Smm}(h')) \end{array}$$

*Proof.* By enumeration of $ev \in \texttt{Event}$ and $h \in \texttt{Archi}_{Smm}$, we check that each requirement introduced previously is preserved by $\Delta_{Smm}$. We use those intermediary results to conclude on $inv_{Smm}$.

Most of the proof work concerns the memory accesses. In case of the uncacheable strategy, the proof is straightforward, as only the system memory is concerned. With the writeback strategy, cache and memory often need to synchronize, which makes the proof tedious. However, the two predicates *cache_clean* and *smram_code* together allow their mutual preservation in the result state, as long as the predicates *valid_smrr* (to protect the integrity of the SMRAM copy within the cache) and *locked_smramc* (to protect the SMRAM integrity) hold in the initial state.

The *smram_pc* predicate is preserved thanks to the *smm_behavior* definition and the *valid_smbase* predicate, the preservation of the latter being trivial as only `smm` can update the SMBASE register and it is prevented by *smm_behavior*. The same approach is used to tackle the *valid_smrr* predicate.

Finally, the preservation of the *locked_smramc* predicate is trivial to prove because it becomes read-only as soon as it is set, so there is no such things as an "unlock SMRAMC" state-transformation.

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.

**Lemma 2 (Second Law).** $\forall h, h' \in \texttt{Archi}_{\mathcal{S}}, \forall ev \in \texttt{Event}_{Soft}$,

$$smm\_context(h) = \texttt{os} \Rightarrow behavior(h, ev)$$

*Proof.* By definition of $behavior_{Smm}$, $smm\_context(h) = \texttt{smm}$ is an antecedent of the conditional.

*$\Delta_{Smm}$ is sound* 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.

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

We prove that this predicate holds true for a state-transformation with respect to the HSE mechanism.

**Lemma 3 (Invariants Enforce Security).** $\forall h, h' \in \texttt{Archi}_{\mathcal{S}}, \forall ev \in \texttt{Event}$,

$$\begin{aligned} &inv_{Smm}(h) \\ &\quad \Rightarrow h \xrightarrow[\textsc{Smmx86}(ctx)]{ev} h' \\ &\qquad \Rightarrow (ev \in \texttt{Event}_{Soft} \Rightarrow behavior_{Smm}(h, ev) \\ &\qquad\quad \Rightarrow smm\_secure\_transformation(h, ev) \end{aligned}$$

*Proof.* By enumeration of $ev \in \texttt{Event}$ and $h \in \texttt{Archi}_{\mathcal{S}}$. By definition of the $smm\_secure$ and $software\_tampering$ predicates and $minx86\_fetched$, the only case which is not trivial is $Fetch$. The $smram\_pc$ invariant forces the value of the program counter register to be within the SMRAM and the $smram\_code$ and $cache\_clean$ invariants force the content of the DRAM memory and of the cache lines to be owned by the SMM code. This prevents the OS code to tamper with the execution of the SMM code.

Finally, we prove the HSE mechanism soundness.

**Theorem 1 ($\Delta_{Smm}$ is Sound).**

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

*Proof.* The "Invariants Enforce Security" lemma applies for one transition and the "Invariants Are Preserved" lemma allows to reason by induction on runs.

Therefore, the HSE mechanism we have modeled succeeds to enforce the SMM Isolation Policy.

## 5 Discussions

As we said, the SpecCert formalism, our x86 model MINX86 and the specification and verification of the SMM HSE have been implemented in the Coq proof assistant system (version 8.5). The project is about 4 500 lines of code (LoC), 190 definitions and 150 proofs (theorems and lemmas).

The model introduced in Section 3 has been implemented in the *x86* module (1 700 LoC). In addition to the hardware architecture states and events, it contains several generic functions to manipulate the states and proofs to verify the implementation of their functions. The module *Smm* implements the SMM HSE specification and verification commented in Section 4.

One way to evaluate our work is to challenge it against disclosed attacks which were able to defeat real life HSE strategies. We take the example of the SMRAM cache poisoning attack [16,21]. This vulnerabilily has been disclosed in 2008 and led to the introduction of the SMRRs. The attack flow is the following: if an attacker can set the proper cache strategy for the SMRAM, then she is able to tamper with the SMRAM content within the cache. The SMRR prevents this scenario because it makes the SMRAM uncacheable for unprivileged software components. We have been able to prove the existence of the SMRAM cache poisoning attack in our model. More precisely we have been able to prove the following statement:

```
Theorem cache_poisonning_attack:
  ∀ h :Architecture,
    ¬smrr_inv(h)
    → ∃ ρ :Execution,
        init(ρ) = h
        → run_of x86Smm ρ
        → ¬run_of x86SecSmm ρ.
```

The SMRAM cache poisoning is not the only attack which has defeated the SMM isolation HSE strategy. Our x86 model is not complete enough and does not take into account the related hardware functions. We would rather evaluate the amount of work which is needed to extend our model to express them.

The Invisible Things Lab, during their work on Xen security, has found a way to use a x86 memory remapping mechanism in order to let an OS circumvent most of the legitimate memory protection mechanisms [22]. It can be used to directly tamper with the SMRAM, even if it has been locked. As one of the invariants of the SMM isolation HSE strategy is the integrity of the SMRAM, introducing the remapping feature within our model of the Memory Controller would break our assumptions. An identical reasoning can be applied to Sinkhole [6], another attack which takes advantage of the APIC registers remapping feature to tamper with the SMRAM.

Kallenberg *et al.* [13] have found a race-condition which allows an attacker to tamper with the SPI Flash. To take into account this attack, we would have to extend our x86 model to add another CPU. Such an improvement, contrary to the previous ones, is far less minor. One way to do it is to assign an identifier to each processor unit and to modify the software events definition accordingly. The security policy should also be updated to take into account the integrity of the SPI Flash.

## 6 Related Works

With SpecCert, we intend to specify and verify HSE mechanisms against reusable hardware architectures models. We have initiated the development of one of these models with Minx86, a minimal model for x86 hardware architectures.

Several x86 models have already been defined. For instance, Greg Morrisett *et al.* have developed RockSalt [19], a sandboxing policy checker, upon such a model. Peter Sewell *et al.* have proposed a model for x86 multiprocessors [24] which aims at replacing informal Intel and AMD specifications. Andrew Kennedy *et al.* have developed an assembler in Coq [14] 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 is a HSE mechanism and we believe we could obtain similar results with 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 (Mur$\varphi$) to model and verify the eXecute Only Memory (XOM) architecture [15]. The XOM architecture allows an application to run in a secure compartment wherein its data are pro-

tected against other applications and even a malicious operating system. The main difference with our approach is that the XOM security properties are enforced by a secure microprocessor without the need for a software component to configure it or other hardware devices. On the contrary, we intend to specify one way to use a set of hardware functions to enforce a security policy.

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) [17,18], 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 article, we have presented a threefold contribution. First, we have proposed a formalism to specify and verify Hardware-based Enforcement Security (HSE) mechanisms against hardware architecutre models. Then, we have defined a minimalist x86 model called Minx86. Finally, we have specified and verified the SMM HSE mechanism against this model. In its current implementation, the Minx86 has a heavy focus on SMM-related features. 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.

We want this work to be a first step towards HSE verification. For now, our proofs are built against an abstract model of the hardware architecture. Ultimately, we aim to extend those 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 [7,17,18] 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. Intel: CHIPSEC: Platform Security Assessment Framework. `http://github.com/chipsec/chipsec`
9. Intel: Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family
10. Intel: Intel 5100 Memory Controller Hub Chipset
11. Intel: Intel 64 and IA32 Architectures Software Developer Manual
12. Intel: Intel Trusted Execution Technology (Intel TXT) (07 2015)
13. Kallenberg, C., Wojtczuk, R.: Speed racer: Exploiting an intel flash protection race condition
14. 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)
15. 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)
16. Loic Duflot, Olivier Levillain, Benjamin Morin, Olivier Grumelard: Getting into the SMRAM: SMM reloaded CanSecWest
17. 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)
18. Makris, Y.: Trusted module acquisition through proof-carrying hardware intellectual property. Tech. rep. (2015)
19. 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)
20. Rafal Wojtczuk, Joanna Rutkowska: Attacking intel TXT via SINIT code execution hijacking
21. Rafal Wojtczuk, Joanna Rutkowska: Attacking SMM memory via intel CPU cache poisoning
22. Rutkowska, J., Wojtczuk, R.: Preventing and detecting xen hypervisor subversions. Blackhat Briefings USA (2008)
23. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) 3(1), 30–50 (2000)
24. 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)
25. Yuriy Bulygin, John Loucaides, Andrew Furtak, Oleksandr Bazhaniuk, Alexander Matrosov: Summary of Attacks Against BIOS and Secure Boot, def Con 22