HAL
open science

Design and Implementation of a Hardware Assisted Security Architecture for Software Integrity Monitoring

Benoît Morgan, Eric Alata, Vincent Nicomette, Mohamed Kaâniche, Guillaume Averlant

To cite this version:

Benoît Morgan, Eric Alata, Vincent Nicomette, Mohamed Kaâniche, Guillaume Averlant. Design and Implementation of a Hardware Assisted Security Architecture for Software Integrity Monitoring. The 21st IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2015), Nov 2015, Zhangjiajie, China. 10p., 10.1109/PRDC.2015.46 . hal-01322882

HAL Id: hal-01322882

https://hal.science/hal-01322882v1

Submitted on 28 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design and implementation of a hardware assisted security architecture for software integrity monitoring

Benoît Morgan *†‡, Éric Alata*†‡, Vincent Nicomette*†‡, Mohamed Kaâniche* and Guillaume Averlant*†‡
* LAAS-CNRS, 7 av. du Colonel Roche, 31400 Toulouse, France
† INSA Toulouse, 135 av. de Rangueil, 31400 Toulouse, France
‡ Université Toulouse - Paul Sabatier, 118 route de Narbonne, 31400 Toulouse, France
firstname.lastname@laas.fr

***Abstract*—The increasing complexity of software and hardware layers makes them likely to include vulnerabilities. Recent research has shown that subtle attacks are able to successfully exploit (through compromised peripherals performing DMA attacks for instance) vulnerabilities in low-level software, even running in the most privileged mode of the processors. Therefore, the security of such systems should not be solely based on components running on the processor. This paper describes the design and the implementation of a security architecture that is designed to securely execute integrity checks of any software running on top of this architecture. It is composed of a security hypervisor, running in the most privileged level of the processor, assisted by a trusted hardware component, autonomous and independent of the processor, regularly checking the integrity of the security hypervisor itself. The design, the implementation of this security architecture, as well as experiments showing the relevance of our approach, are detailed in this paper.**

I. Introduction

Nowadays, computer systems are required to execute more and more functionalities. This trend is confirmed by the existence of increasingly complex operating systems and virtual machine managers. Current operating systems are able to execute many software components in parallel and virtual machine managers are able to virtualize the hardware in order to facilitate the coexistence of different operating systems. Execution of these software components is supported by hardware platforms. Nowadays, a large part of hardware platforms are using x86 processors and PCI Express bus to interconnect their components. These architectures are complex and evolve quickly. The hardware platform is configured at boot time by low-level software located in the mainboard, so-called *BIOS* (Basic Input/Output System). After the configuration of hardware platform, the BIOS hands over to the kernel of the operating system or to the virtual machines manager. The purpose of the kernel and the virtual machines manager is to share and abstract hardware resources between user processes or virtual machines. The virtual machine starts its execution in the same way as a physical one. So, an architecture designed with a virtual machines manager is slightly more complex. The software components running on top of the operating system have to rely on this hardware and software stack on which it is executed. This stack organization also reflects the privileges that each component of a layer possesses.

Hardware components, BIOS, operating systems and virtual machines managers are complex. They potentially contain hundred thousand source lines of code. Also, they are sometimes proprietary like VMWare ESXi VMM. Bugs are possibly present anywhere in the hardware and software stack, sometimes opening exploitable security flaws, even if significant efforts are made to reduce them. The malicious exploitation of these bugs can lead to undesired user software modification or corruption, altering application behavior and expected services. Some security mechanisms have been proposed in order to protect the low level software, such as Trusted Platform Modules or the Intel Trusted Execution Technology [9] which checks the integrity of loaded software at boot time. However, this approach is not sufficient because these mechanisms cannot guarantee the integrity of software at runtime. For that purpose, some runtime mechanisms must be designed so that the software integrity can be preserved, despite the existence of components that would have been compromised.

Recent trend consists in adding privileged protection, based on software or hardware mechanisms (eg. ARM TrustZone or Intel SGX). As described later in related work, these approaches are not sufficient because they solely trust software running on the processor as well as the processor itself, and this software or even the processor can be faulty. To address this issue, we present in this paper the design and implementation of a hardware assisted trusted architecture, so-called *security architecture* in which it is possible to securely execute integrity checks of any software running on top of this architecture. This architecture is a hybrid mix up of software and hardware components. The hardware component is autonomous, totally independent, and cannot be corrupted by any software running on top of the processor. Its design and implementation are described in the paper, as well as preliminary experiments showing the relevance of the approach.

This paper is structured as follows. Section II describes the threat model and the assumptions in our approach. Then, section III presents an overview of the proposed approach. Technical background, useful to understand our architecture,

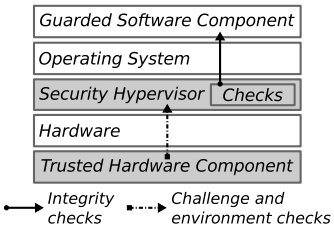

Fig. 1: Security architecture

is given in section IV before the overall presentation of the architecture in section V. Sections VI and VII, provide details regarding the software and hardware components of our security architecture. A discussion regarding the efficiency of the attack mitigation in our architecture is then proposed in section VIII. Some preliminary experimentation results and proof of concepts are provided in section IX. Section X compares our solution to others. Finally, section XI concludes this paper and proposes future work.

II. Threat model and assumptions

This paper addresses software integrity checks at runtime, and proposes the design and implementation of an architecture aiming at providing a secure integrity check execution architecture. We assume that attackers do not have any physical access to every component of the architecture (*assumption 1*). This assumption seems perfectly realistic in the sense that if an attacker gains a physical access to a machine, then he is able to perform worst attacks on the system than simply corrupt software integrity. We assume that attacks are performed through the execution of malicious software on the CPU (*assumption 2*). Moreoever, even if the attacker performs software-based attacks only, we consider that he is able to target software but also hardware components (*assumption 3*). In particular, an attacker is able to reconfigure hardware to use it at his convenience, if the hardware directly offers the facility to reconfigure itself or includes a vulnerability which enables this facility.

According to these assumptions, the threat model is as follows. Malicious software may run at a privilege level higher, equal or lower than the software it tries to corrupt. To perform its attack, it is able to bounce on any other software with a privilege level also higher, equal or lower than its own level. In the same way, it can bounce on hardware components. As an example, an application could exploit some system call API vulnerabilities to increase its privileges. A malware injected into the kernel space has access to the whole physical memory space, and so has enough privileges to modify every software running on the machine. There is also the case of misconfigured COTS software and hardware components, which may expose security flaws because of the nature of their default or misconfigured state (web frameworks, firewalls, database management systems, etc.).

III. Overview of the proposed approach

We consider a black box approach solution in which the developer provides the software component to execute (for instance, a kernel module, a module of the BIOS, a web server, etc.), accompanied by the set of integrity functions dedicated to check this software component (figure 1). In the rest of the paper, this software component is called *guarded software component*. The functions executed in order to evaluate the integrity of this component are called *integrity checks*.

Our trusted architecture is composed of two components. The first one is a *security hypervisor*. The set of integrity checks, provided by the developer, is embedded into this hypervisor which is in charge of executing them. If the guarded software component to check is itself a virtual machines manager, our hypervisor is able to activate the nested virtualization to handle this situation. This security hypervisor runs in the most privileged level of the processor. It is specifically designed and implemented to securely execute these integrity checks. However, this component may be corrupted, for instance by exploiting a misconfiguration of some native COTS hardware components included in the host, that could be abused by a malware. This is why it is necessary to check the integrity of this privileged component itself, even if it runs in the most privileged level of the processor. The only way to detect when this privileged component is compromised is to check its integrity from an external processor directly connected to the system but independent of its operating logic and behavior. As a consequence, our architecture includes a second component, the *trusted hardware component*, that is physically embedded in the host and that is directly connected to the PCI Express bus. This trusted hardware component is in charge of regularly evaluating the integrity of the hypervisor. For that purpose, it must check both the correct behaviour of the security hypervisor as well as the correct configuration of some critical hardware and software components of its execution environment. More precisely, the trusted hardware component regularly proposes *challenges* to be executed by the security hypervisor, and checks the results of the execution of these challenges. It is also able to run several tests in order to 1) detect any alteration in the code or data of the security hypervisor; 2) detect any changes in the configuration of some components of the security hypervisor environment. These checks are so-called *environment checks*. A detection of integrity alteration of the guarded software or of the security hypervisor or its environment triggers an alert. Alerts are handled by a separate trusted machine, through a dedicated

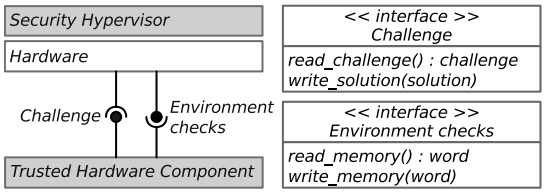

Fig. 2: Security architecture hardware interfaces

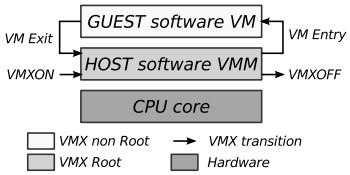

Fig. 3: VMX operations

communication protocol.

The trusted hardware component interfaces must be designed to prevent any malicious exploitation from a corrupted security hypervisor and consequently has to remain simple (figure 2). An interface is provided by the trusted hardware component and allows two interactions: reading the code of the challenge from a dedicated internal memory; writing the solution to a dedicated internal memory. They must have no impact on the trusted hardware component internal behavior. In addition, the hardware and the security hypervisor are also able to read and write anywhere in the physical memory (security hypervisor, peripheral memories).

Each challenge consists in the execution of non trivial operations that compute a value, so-called "the solution" of the challenge. In order to prevent an attacker to precompute or bypass the execution of the challenge to get the correct solution considering both the computed values and the execution time, they must be designed carefully. The following requirement are enforced in our approach:

1) Each challenge is used only once.
2) Each challenge is composed of a set of atomic instructions whose execution times bounds are known.

Let us note that our architecture is especially designed for critical systems security. It is not intented to be used on a personal laptop or PC at home. We made the choice of using a x86 based system with PCI Express interconnected peripherals because our solution must be easily integrated into already existing architectures. Anyway, the principles of our approach are still valid for any architecture with interconnected components.

Before presenting in details this architecture as well as its two components in sections V, VI and VII, the next section is dedicated to some technical background useful to understand the implementation of the components.

IV. Technical background

This section presents a short review of hardware-assisted virtualization technologies used in our solution.

VT-x [8], standing for Virtual Technology, brings hardware support for x86 virtualization in Intel microprocessors through a new instruction set called VMX, a new memory virtualization layer and new execution modes called *VMX root operation* and *VMX non-root operation*.

Virtual Machines (VMs or guests) are executed on virtual cores configured with special processor internal structures called Virtual Machine Control Structures (VMCSs). Virtual machines are executed in *VMX non-root operation* mode, which means that they are under the control of the software entity called the Virtual Machine Monitor (VMM or host). VMs are started with the `vmlaunch` instruction. The VMM gets control on the VM through virtual machine interruptions. These interruptions are configured within the VMCS. After having correctly handled the VM Exit, in the nominal case, the VMM gives the control back to VM thanks to the `vmresume` VMX instruction. Transitions from the VM to the VMM are called *VM Exits* whereas those from the VMM to the VM are called *VM Entries* (figure 3).

The VMM protects its memory space and isolates memory regions of the virtual machines using a so-called Extended Page Tables (EPT). EPT principle and configuration is nearly identical to paging. With EPT, VM physical addresses called guest physical addresses are translated to host physical addresses used by the memory controller.

Just like a system that makes a software believe that it is the only software running on top of the processor, a virtual machine manager may also have to make another virtual machine manager believe that it is the only virtual machine manager installed on the system. This technique is called nested virtualization [5].

V. Global Architecture

This section is dedicated to the overall presentation of our security architecture.

A. Hardware infrastructure

The hardware architecture of our security architecture is depicted in figure 4. It is composed of two x86 machines (the target machine and the trusted remote machine) and a PCI Express peripheral (the trusted hardware component).

The target machine executes the guarded software component and the security hypervisor. It embeds a recent processor including the hardware virtualization assistance technology. It also includes an Ethernet link (Ethernet 2), which is under control of the security hypervisor. This link is considered trusted as long as the security hypervisor is not corrupted. Results of the integrity checks of the guarded software component are transmitted to the remote trusted machine through this link.

A ML605 Xilinx board is used to implement our trusted hardware component. It is plugged on the target machine with a PCI Express riser and an onboard PCI Express 8x connector. The peripheral is connected to a trusted local network to the trusted machine through its Ethernet interface (Ethernet 1). The trusted hardware component provides, thanks to its

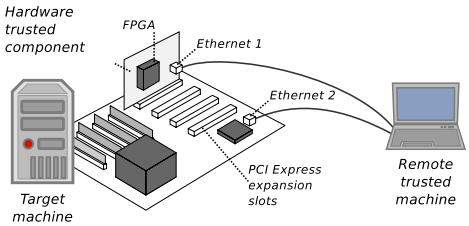

Fig. 4: Security architecture platform

hardware registers, the challenge interface for the security hypervisor. This interface does not provide the hardware capability to configure the peripheral behaviour from the CPU. Furthermore, using this interface does not affect the nominal behavior of the internal cores responsible for the assessment of integrity. The peripheral is also able to execute the environment checks (see section VI for further implementation details). Collected results are checked and integrity status is sent to the remote trusted machine on the Ethernet trusted link.

Finally, the remote trusted machine runs on a COTS operating system. It collects the results of the challenges environment checks, as well as the results of integrity checks, respectively performed on the security hypervisor and on the guarded software component. This remote trusted machine also generates the challenges, the environment checks and makes them available to the trusted hardware component through a TFTP local server.

B. Overview of the integrity checking cycle

There are three phases in the integrity checking cycle. Phase 1 corresponds to the test of the security architecture from the trusted hardware component through challenges and environment checks. Phase 2 is dedicated to the execution of integrity checks of the guarded software component (figure 5). The execution frequency of the integrity checking cycle varies. The waiting time between two occurrences is the phase 3).

Phase 1 includes two different methods.

In the first method (Phase 1.a), the trusted hardware component submits a challenge to the security hypervisor in order to check its behavior with respect to the expected execution time and expected response to the challenge. The challenge is randomly generated to ensure freshness and unpredictability. Each challenge is associated to an expected execution time (see subsection VI-D). The security hypervisor has to execute it and respond within this delay. The response of the execution is collected and verified in the trusted hardware component itself. The delay is controlled by a peripheral internal timer, totally dissociated from CPU clock. This way, maliciously manipulating internal processor timers like the timestamp counter, or performance counters does not affect the detection mechanism. This step ensures the dectection of a malware that would move our security hypervisor in a less privileged level while installing itself into the most privileged level. Challenges must be designed in such a way that they require more execution time if they are executed in less privileged layer or if they are interpreted. Some examples of challenges could be: computing a hash of the results of the *cpuid* instruction for the whole possible parameters[1]; computing the hash of every field of a VMCS and the VMCS pointer, etc.

The second method (Phase 1.b) performs environment checks on the data and the code of the security hypervisor as well as the configuration of some components that belong to the environment of the security hypervisor. These checks

[1]VT-x technology requires the execution of *cpuid* instruction from a less privileged layer generates an exception. Handling this interruption therefore considerably increases the execution time of the instruction.

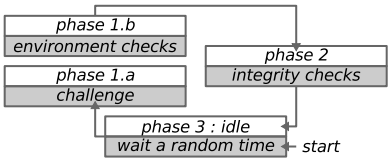

Fig. 5: Integrity checking cycle

are important because a section of the code or the data of the security hypervisor may have been maliciously changed or an essential component for the proper execution of the security hypervisor may have been altered. These attacks may have no perceivable impact on the execution of challenges, letting the trusted hardware component believe that the security hypervisor is non faulty. But, these alterations can be identified by checking the code and the data of the security hypervisor as well as the hardware registers of essential components. For performance reasons, these environment checks are implemented in a dedicated co-processor, the Fast Automata Core (FAC), included in the trusted hardware component. This co-processor is able to execute optimized memory checks, directly hardcoded into the processor. These checks are designed with a dedicated language and a compiler that translates these checks into an automata directly executed by the co-processor. For example, the peripheral is able to read and check, the integrity of the security hypervisor memory space (code, data, EPT pages), but also other peripheral's important memory mapped registers (Ethernet card configuration registers).

Phase 2 consists of the execution of the integrity checks of the guarded software component by the security hypervisor. The integrity checks are carried out through the execution of integrity functions. We consider that these functions are provided by the designers of the guarded software component and are out of the scope of this paper. The security hypervisor must provide a security architecture in order to execute them and send the results of these checks to the remote trusted machine.

Finally, the whole architecture waits for the next cycle triggered by the trusted hardware component at the end of the phase 3.

The internals of the security hypervisor are described in Section VII, while the internals of the trusted hardware component are described in the next section.

VI. Trusted hardware component

Our work is based on the adaptation of the Milkimyst project [6]. The Milkimyst is an open source hardware and software System On Chip (SOC). It is dedicated to video synthesis. The source code of the project is written in C, GNU assembly and verilog for Xilinx FPGAs and it is based on a GNU/Linux distribution. With this design, a user can develop a high-level software intended to be executed directly within the FPGA. However, unneeded functionalities of the Milkimyst SOC have been removed to optimize the FPGA space. For instance, the GNU/Linux distribution has been removed from the SOC because our platform does not need

to execute complex software applications. Our implementation contains the following cores (Figure 6):

- a LatticeMico32 (LM32) micro-processor, [21], 32 bits big endian Microprocessor;
- a Fast Automata Core, so-called hereafter FAC;
- an Onchip ROM;
- a PCI Express Endpoint implementation, so-called hereafter PCIEE;
- an Ethernet MAC;
- a Configuration and Status Register bridge, the CSR bridge and
- a Fast Memory Link bridge to a DDR3 SDRAM controller, the FML bridge.

These cores are interconnected with three different buses: the wishbone [15] bus for memory accesses (CPU data, CPU instructions); the Control and Status Register bus (CSR) [6] for exposing cores registers and the Fast Memory Link (FML) [6] dedicated to SDRAM access and DMA. Let us note that SOC cores can be connected to more than one bus, like the Ethernet that uses the whishbone bus to send or receive frames, and the CSR bus to expose its control and status registers to external components.

A. Protocol phase 1: challenge and environment checks

The architecture has been designed in order to challenge and check the integrity of the security hypervisor from the trusted hardware component. In the following, we introduce step by step how these challenges are executed. The main components involved in each step are indicated between parenthesis after the step title. In the following, TM and SH respectively stand for trusted machine and security hypervisor.

1) Boot: Before entering in the challenge calculation and verification phase, the trusted hardware component needs to boot and configure itself.

Step 1: Boot (LM32): when the board is powered on, the LM32 executes a code within the ROM at an address configured during the SOC synthesis. This code is a basic firmware which configures low level components (the UART, the Ethernet MAC, etc.).

Step 2: Netboot (LM32, TM): once the board is basically initialized, the basic firmware executes a TFTP netbootloader to download a more complex firmware. This firmware configures remaining components of the board (uptime system timer, etc.). It also implements the challenges PCI Express service.

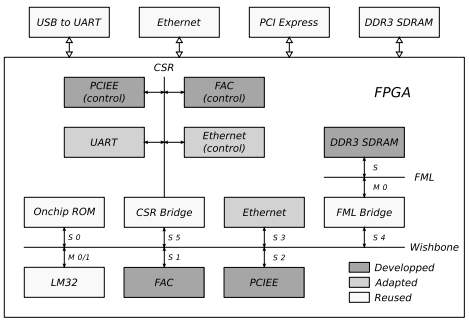

Fig. 6: System on chip

2) Phase 1.a: Challenge: The challenge operating mode implements the PCI Express side of the security architecture challenge execution. It is composed of the following steps:

Step 1: Challenge download (TM, LM32): the firmware downloads the next challenge via TFTP. This challenge is stored in memory of the LM32 for preprocessing. It must not be sent directly to the security hypervisor because it contains the solution.

Step 2: Challenge preprocessing (LM32, PCIEE): the downloaded challenge is preprocessed (cf. section VI-B) in order to extract the solution (expected time and value). The x86 binary VMM code included in this challenge is copied into a PCIEE internal memory. This memory is exposed as a PCI Expansion ROM which is mapped in the security hypervisor memory space through PCI Express.

Step 3: Challenge notification (LM32, PCIEE, SH): the trusted hardware component sends an interruption to the CPU, notifying it that a new challenge is available. The security hypervisor has now a fixed delay of time to download and execute the challenge. For that purpose, it has to make a read access to the expansion ROM. This access is detected by the trusted hardware component thanks to PCIEE events and the system timer is activated.

Step 4: Waiting for challenge solution result (SH, PCIEE): the security hypervisor executes the challenge. At the end of this execution, the security hypervisor must write the solution within a dedicated memory of the hardware trusted component (PCI BARs). This event is tracked by the trusted hardware component using PCIEE events.

Step 5: Solution check (LM32, PCIEE, Ethernet): the solution is written, by the security hypervisor, in the dedicated memory of the PCIEE. The date of this event is compared with the date of the read event of the step 4. If this period duration is significantly different than the one expected, the security hypervisor is considered compromised. Otherwise, the solution provided by the security hypervisor is compared to the solution backed-up during the preprocessing. The remote trusted machine is then notified of the result of the challenge.

3) Phase 1.b: Environment checks: Challenges are not the only way to check the integrity of the security hypervisor. The hardware trusted component is also able to check the integrity of the memory space of the security hypervisor or the integrity of the values of low-level structures (VMCS, page tables, hardware component configuration, etc.). These environment checks concern values that a malware must change in order to corrupt the security hypervisor. So, these integrity checks are adapted for the trusted hardware component to check the integrity of our security hypervisor. In addition, the experiments show that a small amount of time is enough to perform these tests.

Step 1: Automata download (TM, LM32, FAC): the automata intended to be executed on the FAC is downloaded via TFTP. The LM32 copies this automata in a memory directly connected to the wishbone bus and the FAC instruction bus.

Step 2: Page download (LM32, PCIEE): the memory page to be checked is configured and a PCI Express DMA access is requested to the PCIEE.

Step 3: Page Checks (LM32, PCIEE, FAC, Ethernet): once the page is downloaded, according to the type of the page (security hypervisor code, x86 CPU configuration structures, data, etc.), the corresponding automata is executed and the corresponding result is sent to the remote trusted machine.

4) Phase 3: Idle: The trusted hardware component waits during a randomly sampled amount of time until the next cycle.

In the following we provide further explanations about the challenges. More details about the internals of some sub-components of the trusted hardware component can be found in [14].

B. Challenges

A challenge is an algorithm which is composed of a set of atomic instructions whose execution times bounds are known. Every challenge algorithm computes a value, so-called "the solution" of the challenge, from input data. Input data can be characteristic of the integrity of the security hypervisor (control registers, stack pointers or page entries). Challenges are used only once and randomly generated by the remote trusted machine.

An important issue regarding the challenge design is the estimation of its *expected execution time*. To do so, the challenge is firstly executed offline on the same processor as the processor of the security hypervisor, with the timestamp counter, read before and after, to get its measured execution time (called *rated execution time*). We add to the computed time an extra arbitrary delay, to cover the PCI Express message propagation time and so obtain the expected execution time published with the challenge. Furthermore, the challenge is again executed offline, but virtualised with our security hypervisor, to get the *virtualised rated execution time*. The challenge must be designed in such a way that the virtualised rated execution time must be at least 3 or 4 time higher than the expected execution time. This difference allows our system to detect emulation or virtualization of our security hypervisor. To get such a difference, we choose specific instructions for the challenge, hard to emulate and / or which will produce unconditionnal VM Exits (cpuid, vmread, vmptrst). PCI Express physical bus and protocol stack may lead to flow variations due to an overload of the bus, that could invalidate the usage of an arbitrary extra delay in the computing of the expected execution time for a challenge. To face this, PCI Express specification proposes QoS mechanisms which enable to segregate PCI Express messages in traffic classes, ordering them by priority. By using the highest priority class for the messages exchanged by the trusted hardware component, we have additional guarantees that its messages are exchanged in a limited delay, even if the bus is overloaded by traffic class with lower priority. Finally, even if an attacker is able to perform DoS attacks at the physical level, generating artificial jitter after having gained the control on the transcievers of a vulnerable device for example, this attack will be detected by the trusted hardware component which will not receive the responses of the challenge within the expected delay. Finally, in the unlikely case where the natural jitter is too important, generating too much delay in the transmissions, it will result in an false positive alert. Let us note that in our experiments, we never encountered such a situation.

VII. Security Hypervisor

The security hypervisor is designed to be as small and as simple as possible. To operate as the most privileged software component of the system, it needs to virtualize upper layers, correctly isolated with EPT. It also protects itself from attacks coming from malicious hardware, by controlling *Port-mapped Input/Output* (PIO) and *Memory Mapped Input/Output* MMIO spaces as well as configuring the IOMMU. This hypervisor is dedicated to perform a quite simple task, so its binary footprint remains quite small. The two main functions provided by the security hypervisor are: 1) the execution of integrity checks of the guarded software component and 2) the execution of the challenge, sent by the trusted hardware component.

A. Installation and loading strategy

The security hypervisor is an UEFI runtime driver loaded in the preboot environment. The firmware loader prepares its code and data memory space and marks them as used for the operating system running above. Since the adopted strategy is to limit the code footprint of the hypervisor, after activating *VMX operations* and configuring the hypervisor, the control is immediately given back to the firmware in order to let it boot the machine as if the hypervisor were not present (figure 7).

B. Runtime protection

At runtime, the security hypervisor keeps control on the virtual machine thanks to the VM Exits generated by some privileged instruction execution attempts or by the set of instructions which unconditionally generate VM Exits like the `cpuid` instruction. Most of the time, their execution is allowed, except for modifications of *VMX operations* required configuration. The security hypervisor also forbids accesses to its memory space and the memory space of hardware components it wants to keep the hand on, like the Ethernet card used for remote control. To do this, accesses from the CPU itself are controlled with EPT, configured in identity mapping for all the entire physical address space, with read, write and execute rights enabled, except for the regions that must be protected. Undesired accesses will result in VM Exits, called *EPT violations*. Also, DMA acesses from PCI Express devices are controlled with the DMA remapping capability of

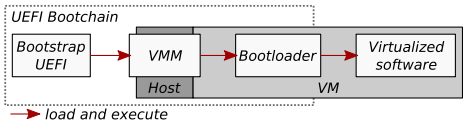

Fig. 7: UEFI Bootchain

Intel VT-d's IOMMU [10], protecting the security hypervisor memory space to be remotely modified. We note that the trusted hardware component accesses are allowed.

VIII. Attack mitigation

Our architecture is a security solution that can also be the target of attacks. This section shows how these attacks are mitigated. To the best of our knowledge, the attacks presented in this section are representative of the different strategies of well-known and documented attacks taken from the state of the art.

A. Full virtualization attack

An attacker could succeed to virtualize our security hypervisor, injecting malicious code in PCI peripheral expansion ROMs and being loaded at the next reboot [7] (assumption 3). Virtualization or emulation of our hypervisor will be detected because the instructions used in the challenge are slower when virtualized or emulated.

B. Relocalization attack

An attacker may be able to exploit a hardware fault or a software vulnerability to relocate the security hypervisor in the memory and install a new hypervisor [11], [27] (assumption 3). The code of the security hypervisor and all the data but the VMCS are left unmodified. The attack will be detected by challenges targetting special CPU internal structures or environment checks. For instance, the challenge can retrieve the location of the hypervisor within VMCS.

C. Direct Memory Access Attack

An attacker may be able to exploit vulnerabilities in PCI Express peripheral to DMA read and write into the security hypervisor memory space [20] (assumption 3). Any malicious modification in the security hypervisor memory space will also be detected by the environment checks.

D. Security hypervisor Ethernet interface

An attacker may also be able to control the point to point Ethernet link (Ethernet 2) to the trusted machine since they are able to gain access to the most privileged level on the CPU (assumption 3). As a consequence, she may send malicious traffic or fake alerts through this Ethernet link. Challenges and environment checks performed by the trusted hardware component will detect this corruption. The trusted hardware component can then warn the trusted machine. Then, this traffic and alerts will be ignored by the trusted machine. Finally, attacks initiated from the trusted machine are not considered in our assumptions (assumption 1 and 2).

E. PCI Express Interface

The trusted hardware component is physically connected to four PCI express lanes. It implements support for read / write in memory mapped registers and read only for its expansion ROM. These two spaces do not influence the behavior of the trusted hardware component internals. Attacker still may try to perform DoS attacks from the CPU or other components to influence the message propagation timing. These attempts will result in a detection by the trusted hardware component itself.

F. Trusted hardware component Ethernet interface

Trusted hardware component Ethernet interface is directly connected to the trusted machine through a dedicated point to point Ethernet link. As a consequence, we do not consider physical attacks on this link. Since the PCI Express interface to the CPU doesn't provide any access to this Ethernet interface, attacks from the CPU are impossible under assumptions 1 and 2.

G. Remote trusted machine threats

The remote trusted machine runs a classical GNU/Linux distribution and is supposed to include the same hardware/-software vulnerabilities and attack vectors of the same class of architectures. That is why this machine is totally isolated from every network during the runtime and why we do not consider attacks towards this trusted machine (assumption 1 and 2).

IX. Experimentations

This section is dedicated to the presentation of validation experiments of our security architecture. In subsection IX-B, some benchmarks experiments are described in order to evaluate the impact on system overall performances. Then, subsection IX-C describes experiments in which corruptions of the guarded software as well as the hypervisor are performed in order to evaluate the detection capability of our architecture.

A. Hardware configuration

The target machine is a Dell precision T1700 with an Intel i7-4770 microprocessor and an Intel c226 chipset. It embeds 8 gigabytes of DDR3 SDRAM. The speed of PCI Express link to the trusted hardware component is 4x. The Ethernet interfaces of the trusted machine (Ethernet 2), trusted hardware component (Ethernet 1) and security hypervisor are connected to a gigabit D-Link DGS-1008D Ethernet switch. The trusted hardware component 8x PCI Express port is plugged in the mainboard thanks to a riser. An additional Ethernet interface of the target machine, used for network benchmarks, is a Broadcom NetXtreme BCM5722 and linked to a 100 megabit non trusted local network.

B. Benchmarks

Three sets of benchmarks have been run on the architecture. Each set is executed on the target machine, firstly without our solution (None column in Table I) and with virtualization only. Then, the full solution is benchmarked, with guarded software integrity checks, environment checks and challenges, which are introduced in section IX-C. Finally the integrity checking cycle is executed every fixed period of time, for benchmarking purpose. Period is changed from 5s to 1s and 100ms. Every time a benchmark is executed, the target machine is rebooted to prevent boundary effects.

TABLE I: Benchmark results

values in seconds

PI	None	Virt.	5s	1s	100ms
Total	221.10	221.37	222.03	223.71	244.89
Average	22.11	22.14	22.20	22.37	24.49
Median	22.11	22.13	22.20	22.36	24.46
Min	22.10	22.12	22.19	22.35	24.43
Max	22.12	22.16	22.25	22.45	24.62
Over.		0.1 %	0.4 %	1.2 %	10 %
Disk	**None**	**Virt.**	**5s**	**1s**	**100ms**
Total	52.21	51.51	52.18	52.23	52.87
Average	5.22	5.15	5.22	5.22	5.29
Median	4.89	4.84	4.85	4.84	4.98
Min	4.64	4.71	4.52	4.68	4.67
Max	8.43	8.14	8.68	8.49	8.70
Over.		-1.3 %	-0.1 %	0.4 %	1.2 %
Network	**None**	**Virt.**	**5s**	**1s**	**100ms**
Total	187.72	187.51	187.92	187.33	186.96
Average	46.93	46.88	46.98	46.83	46.74
Median	46.85	46.89	46.94	46.78	46.69
Min	46.65	46.68	46.61	46.56	46.67
Max	47.36	47.06	47.44	47.21	46.91
Over.		-0.1 %	0.1 %	-0.2 %	-0.4 %

The first set of experiments computes ten million PI decimals with GNU Multi Precision, for 10 iterations. The second set copies, by means of the `dd` Unix command, a 512 megabytes file, on the same disk, in a ext4 partition, 10 times in a row. The third set transfers over SSH a 512 megabytes file from the target machine hard disk to a remote machine in the non trusted local network, 4 times in a row.

Looking at the results, for CPU burn-in test, measured overhead (table I) is less than 0.4 %, for a 5s period and less than 1.2 %, for a 1s period, which is very acceptable. The overhead is rising to 10 percents for a 100ms period because of the nature of the PI benchmark which only consumes CPU. This overhead can be drastically decreased (half of it at least) by optimizing the security hypervisor network communications (stopping the debug mode and context change communication) or optimizing the guarded software integrity checks. Let us note that a 100ms period is maybe too fast and won't increase platform detection accuracy. This experiment is interesting but is not sufficient to estimate the performance of our architecture because a CPU burn-in experiment is not representative of the different use cases of a computer. This is why two other series of experiments are considered, focusing on disk and network usage. These experiments show that performance variations due to the disk accesses latencies and network communication delays are more impacting the system than our architecture. That is why negative and low overhead are measured, which shows that our solution provides very good performances.

C. Corruption detection

In this section, four corruptions (figure 8) are introduced at two different levels, involving two different guarded software, the security hypervisor and the flash memory of the host computer. Guarded software are the `nginx` HTTP server and the `e1000e` Intel driver, associated to their own integrity checks. In both cases, the md5 checksum, calculated during the loading phase of the software, is compared to the checksum calculated at runtime. An environment check is performed on the UEFI platform flash embedded boot order. The boot order is read and backed up as a reference at boot time and then compared to the current one at runtime. The security hypervisor challenge is based on 256 iterations of xor hash computing of the overall `cpuid` instruction parameter space. Expected time for this challenge is thresholded at 512 microseconds.

The considered threat models are malicious boot order modification through Unix user privilege escalation, malicious `nginx` and `e1000e` driver code modification thanks to a malicious linux kernel module and finally, a nested hypervisor rootkit installation and execution virtualizing our security hypervisor.

The corruption has been detected in the four cases. The guarded software memory footprint modifications have been successively detected for `nginx` and `e1000e` driver (1, 2). Then, the security hypervisor has successfully detected the modification of the boot order in the flash memory (3). In the fourth case, because of the malicious virtualization of our security hypervisor, the challenge execution time has been multiplied by 3.6, increased from 321 to 1170 microseconds (4). Let us note that, in this example, a malicious hypervisor, emulating the hardware, can affect the guarded software corruption detection but this will be detected thanks to the challenge.

As a conclusion, the first sets of experiments that we carried out are promising to our viewpoint. The overall performances of a system with and without our security architecture are very close, which shows that our security architecture does not degrade significantly the performance of the system hosting it. Moreover, all corruption attempts that we carried out, either on the guarded software component or the security hypervisor itself, were successfully detected.

X. Related work

Many research works have addressed issues in the domain of software integrity checking protocols.

A first category of integrity checking solutions relies on CPU internal privilege levels mechanisms to separate the monitored code from the monitoring code. To do so, high privileged CPU modes execute the monitoring code while the

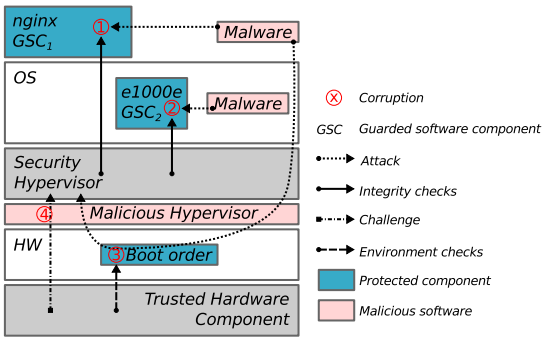

Fig. 8: Corruption detection

monitored code is executed by a less privileged CPU mode. Hytux [12] is a type-2 hypervisor which uses this approach to protect a Linux kernel. This hypervisor has been inspired by Invisible Things Lab works on Blue Pill [19], a tiny type-1 hypervisor rootkit. BMCS [18] is an hypervisor similar to Hytux but implemented on KVM. HIMA [3] is a generic integrity analysis tool for virtual machines, independent of the virtualized operating system, based on Xen [4]. HyperTap [17] is a component integrated in KVM dedicated to the monitoring of virtual machines. It addresses both reliability and security monitoring aspects in a two phases process which are logging and auditing. HyperCheck [25] is an integrity testing tool based on a different approach. It is not implemented as an hypervisor but it is designed as a service for the system machine management mode (SMM mode) and installed in the memory of a network card. Some other works attempt to protect the monitored code with a monitoring code executed with the same privilege level as the monitored code. SIMA [22] is an hypervisor extension which aims to protect the hypervisor against malware installed inside virtual machines. HyperSafe [26] offers an extension of this solution by adding self monitoring through execution flow control of the hypervisor. The main drawback of these solutions is that the monitoring software is not isolated from the protected host. Such isolation is provided by TinyChecker [23], which offers a nested virtualization [5] based approach to protect virtual machines critical data and which can recover the faults of the virtualized hypervisor. It is implemented with Xen hypervisor.

These approaches are insufficient to our viewpoint because they need to trust the CPU or the firmware they use. Copilot [16], a security coprocessor connected as a peripheral, addresses this problem. It offers a completely external integrity checking solution, computing hashes of memory pages owned by an operating system kernel. This approach is the closest to our proposed architecture, but is not sufficient to our point of view because it blindly trusts its environment. Indeed, the following situation may happen: the monitored code is correct but a malicious software has managed to take control of the behaviour of the monitored code without changing it, by being run in a more privileged mode or installed into another place in the memory. Moreover, the semantic gap [13] from a connected peripheral to a kernel or a user application is extremely hard to address. For example, some CPU internal registers are essential to know which software component is installed as the host hypervisor, and the content of such registers is impossible to get from an external peripheral. A software/hardware collaboration based approach is, to our viewpoint, more realistic to execute more complex and efficient challenges. Finally, a hash only based solution is not scalable on servers running several virtual machines with 64 gigabytes of RAM for example.

Processor foundries add more and more privileged layers responding to user requirements. For example, multi tasks systems have been implemented with Memory Management Unit (MMU), segmentation (rings) and protected mode. BIOS developers and mainboards manufacturers are using Intel SMM, to implement highly privileged and secured low level system management code. Hardware assisted virtualization technologies have been also proposed, such as VT-x, AMD-V or ARM virtualization extensions. ARM TrustZone[1], also, is an ARM architecture extension integrated to the third generation of Cortex-A processors. It segregates softwares in two execution domains, the *secure world*, more privileged than the *normal world*. The *secure world* brings a secure software execution environment for tiny security kernel. Disregarding potential hardware bugs, with ARM TrustZone, highly privileged software is efficiently protected from being directly corrupted by lower level software, but it is not protected from the exploitation of vulnerabilities introduced in the code itself, whatever the privileged mode it is being executed, as long as there is an interface to access to the vulnerabilities (system calls, hypercalls). Intel SGX [2] is a set of new CPU instructions that allows software applications to create a private execution environment also called *security enclave* by Intel. The code and data of this enclave are not accessible by any other applications and kernel code. The point is that applications protected by SGX undergo the same problem as with ARM TrustZone, they also interact with their environment, and may be corrupted through these interactions. If the code itself includes vulnerabilities, Intel SGX cannot prevent the exploitation of its vulnerabilities. This doesn't mean that Intel SGX or ARM TrustZone are not compatible with our approach. When SGX will be accessible, we could, for instance, use it to locally enhance the security level of our hypervisor code[2].

In summary, solely adding more and more privileged layers to ensure security properties cannot be sustainable, because the more privileged layers are added, the more malwares have a chance to insert themselves in these layers. As these layers are more and more privileged, consequences of exploitation flaws may be disastrous [11]. To deal with this issue, an hybrid approach, composed by a software security layer, checked by an autonomous trusted peripheral, must be considered to our viewpoint.

XI. Conclusion and future work

In this paper, we have presented the design and the implementation of a trusted security architecture that is aimed at securely executing integrity checks of any software running on top of this architecture. This architecture is composed of a security hypervisor, running in the most privileged mode of the processor, and an external autonomous hardware trusted component, independent of the processor, and checking the integrity of the security hypervisor itself. This trusted architecture was proposed because we argue that installing integrity monitoring software in subsequent or same security layers than a monitored software is not sufficient. Indeed, hardware vulnerabilities or misconfigurations may lead to monitoring software corruption. On the other hand, only external hardware solutions are not sufficient for semantic gap and scalability

[2]Other examples of hybrid approaches are discussed in [24]

reasons. To face these issues, we argue that a mixed software and hardware integrity checks execution architecture is more realistic because it enables to run software integrity checks inside a security hypervisor, while trusting this hypervisor thanks to environment checks and challenges realised from a trusted hardware component. First validation experiments have been carried out and show that CPU performance overhead is very acceptable, and insignificant in some cases. Finally, several examples of attacks leading to the corruption of two guarded softwares have been successfully detected.

Regarding future work, we plan to test our integrity checking solution on a larger panel of guarded softwares like kernel or BIOSes internals as well as virtual machine managers. In particular, the ESXi VMware virtual machine manager is the next software which we plan to adopt as a guarded software component. We also plan to carry out a more intensive test campaign of attacks in order to evaluate the detection capability of our architecture. Also, the trusted hardware component adapted to support multi-core. Another interesting future work in order to enhance this architecture consists in exploring the possible integration in our security architecture of the new features embedded in in most recent processors. In particular, the SGX instructions, announced for the future Intel processors, seem to be a good candidate. Finally, we also plan to test the portability of our solution to other hardware architectures, such as ARM processors.

XII. Acknowledgement

This research is partially funded by the Secured Virtual Cloud project of the French program Investissements d'Avenir on Cloud Computing.

References

[1] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013.

[3] A.M. Azab, Peng Ning, E.C. Sezer, and Xiaolan Zhang. Hima: A hypervisor-based integrity measurement agent. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 461–470, 2009.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[5] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *OSDI*, volume 10, pages 423–436, 2010.

[6] Sebastian Bourdeauducq. *A performance-driven SoC architecture for video synthesis*. Skolan för informatons-och kommunikationsteknik, Kungliga Tekniska högskolan, 2010.

[7] Pierre Chifflier. Uefi et bootkits pci : le danger vient den bas. In *Actes du 11ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pages 159–190, 2013.

[8] Intel Corporation. Intel 64 and ia-32 architectures software developers manual combined volumes:1, 2a, 2b, 2c, 3a, 3b, and 3c. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[9] Intel Corporation. Intel trusted execution technology. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf.

[10] Intel Corporation. Intel virtualization technology for directed i/o. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf.

[11] Loïc Duflot, Daniel Etiemble, and Olivier Grumelard. Using cpu system management mode to circumvent operating system security functions. *CanSecWest/core06*, 2006.

[12] E. Lacombe, V. Nicomette, and Y. Deswarte. A hardware-assisted virtualization based approach on how to protect the kernel space from malicious actions. In *18th EICAR Annual Conference*, pages –18, May 2009.

[13] Lionel Litty, H Andrés Lagar-Cavilla, and David Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, pages 243–258, 2008.

[14] Benoît Morgan, Éric Alata, Vincent Nicomette, Mohamed Kaâniche, and Guillaume Averlant. A hardware-assisted security enclave for software integrity monitoring, 2015.

[15] Silicore Opencores.org. Specification for the: Wishbone system-on-chip (soc) interconnection architecture for portable ip cores. http://cdn.opencores.org/downloads/wbspec_b3.pdf.

[16] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194. San Diego, USA, 2004.

[17] Cuong Pham, Z. Estrada, Phuong Cao, Z. Kalbarczyk, and R.K. Iyer. Reliability and security monitoring of virtual machines using hardware architectural invariants. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 13–24, June 2014.

[18] Wu Qingbo, Wang Chunguang, and Tan Yusong. System monitoring and controlling mechanism based on hypervisor. In *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, pages 549–554, 2009.

[19] Joanna Rutkowska. Introducing blue pill. http://theinvisiblethings.blogspot.fr/2006/06/introducing-blue-pill.html, 2006.

[20] F. Lone Sang, E. Lacombe, V. Nicomette, and Y. Deswarte. Exploiting an i/ommu vulnerability. In *International Conference on Malicious and Unwanted Software (MALWARE 2010)*, pages 9–16, Oct 2010.

[21] Lattice Semiconductor. Latticemico32 processor reference manual. http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/LatticeMico32.aspx.

[22] B. Stelte, R. Koch, and M. Ullmann. Towards integrity measurement in virtualized environments - a hypervisor based sensory integrity measurement architecture (sima). In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 106–112, 2010.

[23] Cheng Tan, Yubin Xia, Haibo Chen, and Binyu Zang. Tinychecker: Transparent protection of vms against hypervisor failures with nested virtualization. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, 2012.

[24] Paulo E Veríssimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1):66–81, 2006.

[25] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, RAID'10, pages 158–177, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395, 2010.

[27] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning. http://invisiblethingslab.com/itl/Resources.html, March 2009.