UNIVERSITY OF PISA
COMPUTER SCIENCE DEPARTMENT

The Prosper runtime monitor: design and formal verification

Candidate:
Hind Chfouka
hind.chf@gmail.com

Supervisors
Andrea Corradini
Roberto Guanciale
Examiner
Fabrizio Baiardi

Academic year 2013-2014

To my sister Ymane

Abstract

Runtime monitoring is a technique that can be used to guarantee a certain security property over a system resource. In computer systems, a common approach is to deploy a runtime monitor as a security module of the Operating System's kernel. This approach suffers from some vulnerabilities that can compromise the integrity of the security module. In a virtualized environment, an alternative approach is to exploit the isolation property to protect the security module. In this thesis, a runtime monitor supported by the Prosper hypervisor is presented. This is a fully verified Virtual Machine Monitor for embedded system targeting an ARM CPU architecture. The runtime monitor presented is able to guarantee a security property thanks to the monitoring of the hypercalls that are provided to the guest Operating System by the hypervisor. Through a formal methodology, the thesis discusses the enforceable security properties for the proposed monitor and identifies a security property that permits the protection of the Operating System from code injection attacks. The enforcement of this property is possible thanks to a validation mechanism of the hypercalls that is formally identified in this work. The thesis presents the proof of correctness of the validation mechanism that is fully verified with the support of the HOL4 proof assistant.

Acknowledgements

I would like to thank my supervisors Andrea Corradini and Roberto Guanciale for the academic support; I express my gratitude to Andrea Corradini for all the feedbacks related to the thesis, for introducing me to the topic of the formal verification and for being my academic tutor during the last two years of my master studies. I am thankful to Roberto Guanciale for the valuable guidances during my studies and research at the Royal Institute Of Technology, thank you for the time spent to discuss the thesis and all my doubts. I would like to thank my examiner Fabrizio Baiardi for the important feedback about the thesis. My thanks goes also to Hamed Nemati and Oliver Schwarz for the technical help provided to me during the project work.

My sincere thanks also goes to Mads Dam and Christian Gehrmann for accepting my Erasmus Placement proposal and letting me take part of the Prosper project.

Besides the academic acknowledgements, I would like to thank Giulia and Roberto, Andrea Azzarà, Christiane Baum for their friendship during the months spent in Stockholm. Also, I would like to thank Aida and Bengt Nilsson for supporting and orienting me in the Swedish society.

I am grateful to my uncle Ahmed thanks to whom my passion for the mathematics started since an early age. My sincere thanks also goes to Maura Casella, thank you for your presence and support throughout my years of studies.

Thanks to my friends and colleges Ilaria Ceppa, Lorenzo Vannucci, Alessandro Bianchi, Laura Guidi, Annalisa Cipolli, Chiara Marcheschi and Marco Ponza for the time spent together. Your friendship coloured my days at the Polo Fibonacci for the last six years :-).

A special thanks goes to my parents. I am grateful for your unconditional love, support and patient. Thank you Ymane for being my sister, my strength and my

inspiration in life. There are no words to express how I feel blessed for being part of our splendid family.

A big thanks goes to Emilio, thank you for your love and support and for being there for me.

Last but not least, I would like to thank my friend Giusi, thank you for being my best friend in life.

Contents

Chapter 1

Introduction

Runtime monitoring [28] [15] [19] is a computing system analysis technique based on the observation of a running system and the consequent reaction in case of violation of certain security properties. A common application domain of this analysis technique are Operating Systems. An OS has the task of managing and protecting the system resources (memory, I/O devices, files, etc) from malicious accesses, based on a security policy. An approach [33] for realizing the system resources protection is to deploy a kernel security module that monitors and controls all the accesses of the system's subjects (processes, applications, users, etc) to the resources. The security module intercepts all the access operations on the basis of a security policy, and validates them using a suitable validation mechanism.

The approach of protecting the system resources with a kernel module suffers from some vulnerabilities [9] that are a motivation for this thesis. The runtime security module that protects the system resources is a security-critical component of the system. Therefore, a malicious manipulation of the module itself can compromise the system resources protection, and have undesirable consequences for the whole system security. This kind of attacks are possible in practice since commodity OS (Linux for instance) are not tamper-resistant [9], which means they are vulnerable to malicious manipulations as shown by many case studies in the literature on OS security.

A kernel security module should be part of the trusted computing base. Therefore, it is desirable to be able to *formally verify* its correctness. Unfortunately, the task of the formal verification of a software component becomes infeasible when it comes to an Operating System's module. In fact, it is well known that the com-

plexity and size of Operating Systems are an obstacle to their formal verification.

In this thesis, we present an alternative approach for the runtime monitoring of a commodity OS that takes advantage of the *virtualization* [26] [31] technology. The approach relies on the *isolation property*, that is one of the main benefits of the virtualization, to provide a tamper-resistant runtime monitor. The runtime monitor proposed is based on the Prosper hypervisor [3] [11] [12] [20]. This is a fully verified Virtual Machine Monitor for embedded systems targeting the ARM architecture [27] [29] and developed in the Prosper project[1].

We present the general design choices for a runtime monitor supported by the Prosper hypervisor. The goal of this monitor is to control the system running as guest of the hypervisor, and to enforce a correct system state with respect to a security property. The design choices exploit the infrastructure provided by the hypervisor and allow the *complete mediation* of the runtime monitor in the operations that are *sensitive* for the security property.

In this work, we present a general analysis that identifies what kind of security properties can be enforced by the Prosper monitor. This analysis takes into account the infrastructure provided by the Prosper hypervisor. Furthermore, since the Prosper monitor belongs to the general class of *Execution Monitors (EM)* [28], the thesis takes into account that the enforceable properties by an EM must satisfy a necessary condition, that is formalized in [28], in order to be ensured.

A significant part of the thesis studies a particular application of the Prosper monitor. This application has the goal of protecting the Linux kernel running as guest of the hypervisor from *code injection attacks*. To this purpose, we identify two security properties that need to be enforced. Through a formal methodology, we define an appropriate validation mechanism for the Prosper monitor that is able to guarantee the security properties. Therefore, we provide a complete proof of correctness for the validation mechanism. This proof is fully verified with the assistance of the HOL4 theorem prover [17].

An important part of the work described in this thesis has been realized at the Theoretical Computer Science group of the Royal Institute of Technology (KTH), where part of the Prosper project is developed.

[1]The Prosper project is a collaboration between the Security Lab of the Swedish Institute of Computer Science, and the Theoretical Computer Science group at the Royal Institute of Technology (KTH)

1.1 Structure of the thesis

This thesis is structured as follows:

- Chapter 2 introduces all the background needed to understand the following chapters. Section 2.1 presents the virtualization describing the main concepts behind this technology, the main types of virtualization and its advantages. Section 2.2 presents the Prosper hypervisor and the formally verified isolation properties. Section 2.3 presents Automated Theorem Proving as a formal method for software and hardware verification, introducing the HOL4 theorem prover that is used to verify the proof of correctness of the validation mechanism.

- Chapter 3 presents the Prosper monitor designed and formally verified in this thesis. Section 3.1 discusses the main motivations of the approach proposed for the Prosper monitor, and describes the goals of the thesis work. Section 3.2 presents the Prosper monitor discussing the principle design choices, the enforceable security properties, and the application to the code injection attacks. Section 3.2.3 discusses the validation mechanism for this application, and Section 3.3 presents the proof of correctness of the validation mechanism.

- Chapter 4 presents the verification of the formal proof with the HOL4. Section 4.1 presents briefly the general structure of this verification. In order to give an idea to the reader about a machine assisted verification with the HOL4, in Section 4.2 we provide an example of proof for a lemma.

- Chapter 5 concludes the work presented in this thesis, mentioning some related works in Section 5.1 and discussing the possible future developments in Section 5.2

Chapter 2

Background

2.1 Virtualization

One of the most efficient approaches for managing complexity in computer systems is their division into *levels of abstraction* [31] [26]. Each level is responsible of managing a set of the system resources at a different level of details, providing a well defined *interface* to the other levels. The levels of abstraction or layers are organized in a hierarchy: lower layers are implemented in hardware and higher layers in software. The use of an interface allows the task of a computer system design to be divided into independent sub-tasks, so that the hardware design and the software design can proceed independently. This is only possible if the design of each part respects the specification of the same instruction set, that is a well defined interface. An Operating System as well provides an abstraction to the applications running on top of it. This is possible thanks to an interface defined as a set of system calls that are used by the applications developers without the need of hardware details knowledge.

The interface based approach has many advantages to manage computer system complexity, but it can be restrictive: a component that is designed for a specific interface will not work with other interfaces. There are different processors with different instruction sets, and there are different operating systems. For example, a user application distributed as binary program is committed to a specific instruction set and operating system. Therefore, diversity in computer system architecture reduces the portability of a computer software component. Another

restriction is related to system resources consideration. A traditional computer systems design relies on the implicit assumption that there is a single operating system responsible of managing the system resources. This assumption reduces the system flexibility in case more operating systems would like to share a single set of resources.

Virtualization is the technology that overcomes the restrictions of flexibility and portability. When a computer system component is virtualized, its interface and all resources visible through the interface are mapped into the interface and the resources of the real system. Thus, with the virtualization a second (virtual) version of the real component is created so that it appears to be different. Virtualization can be applied to a computer system component, such as disks, I/O devices, memory, etc, but also to the entire hardware machine. In this thesis, we are interested in the last type of virtualization, that is in *Virtual Machines*.

2.1.1 Overview on virtualization

Virtualization [26] [31] [7] introduces an additional layer that creates a virtual version of the real computer system or subsystem. The virtual version is different from the real and physical subsystem, but thanks to the additional layer it is possible to emulate the real subsystem behavior. Let's consider the example of a hard disk. In some applications, it may be desirable to partition a single large disk into a number of small virtual disks. Having multiple virtual disks is more flexible than having a single physical disk, for example during resource allocation to user applications. The virtual disks are mapped to a real disk by implementing each of the virtual disks as a single large file on the real disk. It is therefore necessary to introduce an additional layer that provides a mapping between virtual disk contents and real disk contents using the file abstraction as an intermediate step. Furthermore, each read or write operation from the virtual disk (namely an operation on the file) is emulated by the additional layer with a real disk operation, thanks to the mapping created.

More formally, virtualization involves the construction of an homomorphism f that maps a real *host* system to a virtual *guest* system [26]. This homomorphism maps the real state to the guest state, and for each possible sequence of operations that modify the state of the host from S_i to S_j, there is a corresponding sequence of operations that perform an equivalent modification to the guest state from S'_i to S'_j.

It's worth to stress that virtualization is different from abstraction. Virtualization does not necessarily hide details, and the level of detail in a virtual system is often the same as the underlying real system.

The virtualization of interest in this thesis is the *hardware virtualization*.

Hardware virtualization

With hardware virtualization the intent is to create a *Virtual Machine*, that is an efficient and isolated duplicate of the real machine. This virtualization is implemented by adding a layer of software called *Virtual Machine Monitor (VMM)* to the real machine, as illustrated in figure 2.1. The VMM creates a simulated machine environment (the virtual machine) for the software running on top of the real machine. In general a VMM allows the execution of complete operating systems reaching a high versatility of the computer hardware.

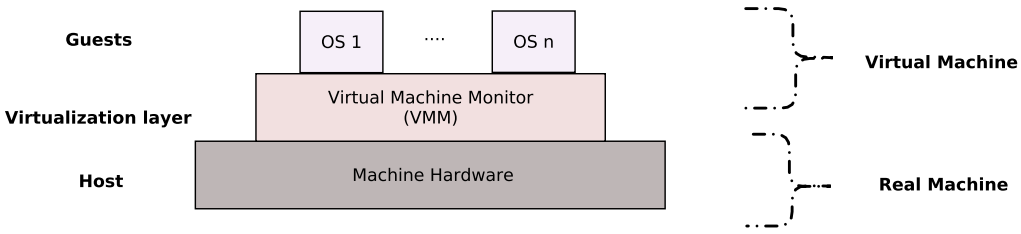

Figure 2.1: Hardware Virtualization

In hardware virtualization, the real machine plays the role of a host, the virtual machine plays the role of the guest, whereas the VMM realizes the homomorphism mentioned above.

The article [26] of Popek and Goldberg is often referred to as the original source for the VMM criteria. Here the conditions for a computer architecture to support virtualization are defined. They also state the conditions a virtual machine monitor must satisfy. These conditions are referred to as *VMM properties*, and they are:

- **Equivalence**: The guest software should exhibit a behaviour that is *essentially identical* to the one demonstrated when running directly on the real

machine. This property is also referred to as *fidelity*.

- **Resource Control**: The VMM must have complete control of the system resources. This property is referred to as *Safety*.

- **Efficiency**: A significant number of machine instructions must be executed without the VMM intermediation or, in other words, by the real machine itself. This is referred to as *Performance*.

In order to satisfy the fidelity property, the environment provided by the VMM must be equivalent to the real machine. This is possible if and only if: for any possible starting state S_1 of the real machine such that $f(S_1) = S_2$, where f is the homomorphism mentioned above, if the real machine reaches a consistent state S_1', then the virtual machine reaches a state S_2' such that $S_2' = f(S_1')$. To fulfil the safety property, the VMM must be invoked for each attempt of affecting the system resources configuration by any arbitrary guest software. Finally, the third property depends on the ability of the VMM to provide a virtual environment in an efficient and optimized way.

Based on [26], the problem to be addressed in the creation of a VMM, when operating within the characteristics of the Instruction Set Architecture (ISA) of the targeted host machine, is the satisfaction of the above properties. Assuming that there are two possible execution modes in a computer system, user and privileged mode, the ISA includes two groups of instructions:

- *Privileged instructions:* these are instructions that trap if the processor is in user mode and do not trap if it is in a privileged mode. An instruction traps when a fault is raised causing a synchronous interrupt that must be handled by the appropriate handler.

- *Sensitive instructions:* these are instructions that change the configuration of the system resources.

The main result in [26] states that the construction of VMM that satisfies the three properties is possible only if each sensitive instruction is privileged. This criterion is now known as *classically virtualizable*. Assuming that a computer system is classically virtualizable, the VMM is supposed to work with each group of instructions maintaining the conditions of equivalence, resource control and

efficiency. To satisfy the first two properties, a VMM manage the guest software and the underlying real machine through *emulation*, *isolation*, *allocation*:

1. **Emulation**: emulation is important for all guest operating systems. The VMM must present a complete real environment, or virtual machine, for each software guest including operating system and user applications. Ideally, the OS and application are completely unaware they are sharing system resources with other applications. Emulation is the key to satisfy the equivalence property. However, in some cases reaching this complete unawareness is not possible. To overcome this problem a special virtualization has been introduced that is called *para-virtualization*. This is explained more in detail in section 2.1.2.

2. **Isolation**: isolation is important for a secure and reliable environment. Each virtual machine should be sufficiently separated and independent from the operations and activities of other virtual machines. Faults that occur in a single virtual machine should not impact others. This property provides high levels of security and availability in a virtualized system.

3. **Allocation**: the VMM must allocate the hardware resources to the virtual machines that it manages. Through allocation, the VMM satisfies the resource control property.

In a classically virtualizable system, a VMM must reside at a higher privilege level than the guest operating system. This is a necessary assumption to fulfil the complete resources control requirement. On the other hand, modern operating systems are designed to run in the highest privilege level. This is a challenging problem to be addressed when designing a hardware virtualization. In fact, without any intervention, a guest operating system is not able to perform privileged instructions to manage the system resources. This problem is solved differently based on the type of virtualization adopted.

2.1.2 Types of virtualization

In the literature on virtualization [32] [31], we can find different macro classes of virtualization. *Server virtualization* is basically another notation for the hardware

virtualization discussed in section 2.1.1, otherwise called *system virtualization*. Another class of virtualization is the *Storage virtualization* that deals with the virtualization of storage devices (hard disks, memory storage, etc..). Network virtualization instead applies the general concept of virtualization to a computer network architecture.

With restriction to the hardware architecture, there are different paradigms of virtualization that are deeply discussed in [13] and [14]. Here a summary is presented:

Full virtualization

As discussed in Section 2.1.1, one requirement for a classically virtualizable system is that the Virtual Machine Monitor must execute in privileged level, while the guest OS executes in a non-privileged level. In this way, the guest OS is not able to perform the privileged operations needed for the resources management, but it must rely on the VMM mediation. In other words, the execution of privileged instructions are delegated to the VMM. In the full virtualization, each time the guest OS tries to execute a privileged instruction this generates a trap to the privileged mode. Since the VMM executes in a privileged mode, one possibility is to have a trap handler that emulates the privileged instruction requested by the guest OS. In this way, privileged instructions are handled by the VMM before their real execution by the real machine, while non-privileged instructions performed by the guest OS do not trigger the trap-and-emulate mechanism.

Advantages: the advantage with this type of virtualization is that the guest OS can run without any modification. This is possible since the virtual interface provided to the guest is identical to the real machine interface. Consequently, binary code of the operating system and user applications can run without any modification.

Disdvantages the trap-and-emulate mechanism introduces a significant overhead for each privileged execution performed in the system. Moreover, full virtualization can be applied only under the classically virtualizable assumption, since if there is a sensitive operation that is not privileged, the trap-and-emulate mechanism is not sufficient to guarantee a complete resource control by the VMM.

Binary translation

This virtualization paradigm is presented as a solution in case the group of sensitive operations is not a subset of the group of privileged operations. In this case, the Popek and Goldberg requirement for a classically virtualizable system is not satisfied, and full virtualization cannot be applied. With binary translation, the idea is to perform a scan of the guest code (dynamically) identifying all sensitive operations, that do not trap, before they are executed. The identified operations are replaced an explicit invocation of the VMM. This approach increases the code size of the guest software introducing more vulnerabilities from a security point of view. In fact, it is desirable to have the amount of code running in privileged mode as small as possible: this reduces the area of possible attacks that affect the security of a computer system.

Advantages: The main advantage of binary translation is making possible the virtualization in case the Popek and Goldberg requirement is not satisfied.

Disadvantages: The scanning and replacement technique introduces an overhead that is larger than the overhead introduced in the full virtualization.

Para-virtualization

Para-virtualization eliminates much of the trap-and-emulate overhead introduced with the VMM implementing the virtualization. But to achieve this improvement the guest operating system must be modified. In para-virtualization, all privileged instructions in OS kernel must be modified to perform the appropriate invocation to the VMM. The guest OS communicates directly with the VMM through an interface provided by the VMM as illustred in figure 2.2. This direct communication eliminates the overhead introduced by the emulation mechanism of the full virtualization paradigm. The calls provided by the VMM interface are invoked by the sensitive operations that manage the system resources (calls for memory configuration and management, for interrupt handling, etc).

Advantages: The VMM interface offers an abstraction that is better than the trap-and-emulate mechanism. In addition, a VMM call performs operations

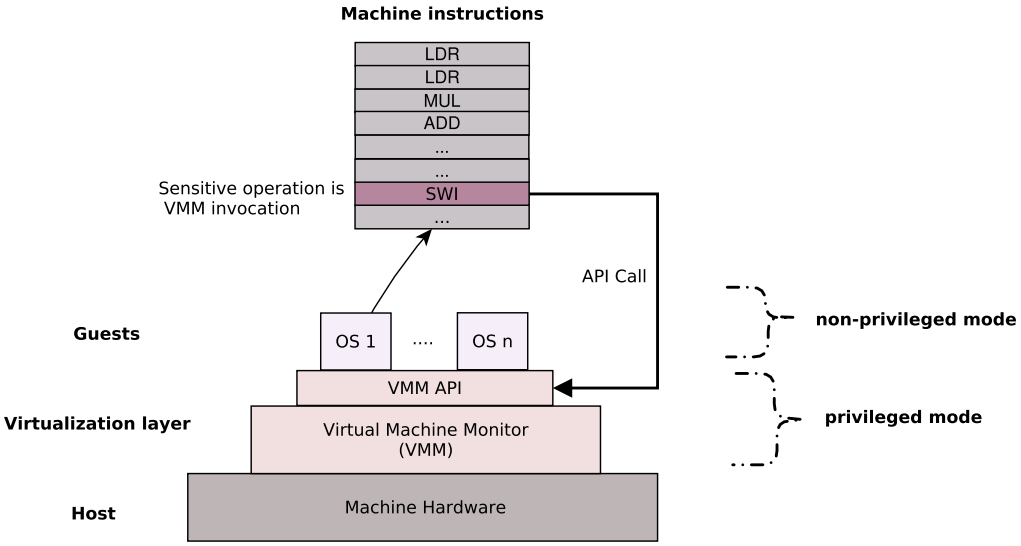

Figure 2.2: Paravirtualization

more efficiently than the emulation of a sensitive instruction. This introduces a significant improvement of efficiency.

Disadvantages: The inconvenient with para-virtualization is that a guest OS must be adapted to use the VMM's interface. This porting can require a lot of work. Besides, a closed-source operating systems cannot be ported to use the VMM interface without a knowledge of its design.

2.1.3 Advantanges of virtualization

Thanks to hardware virtualization it's possible to enable entire virtual machines to be logically separated by the VMM from the hardware they run on. This flexibility creates interesting possibilities for a system design and allows software designers to be independent from the machine architecture characteristics. Virtualization technology enhances other positive features highly required in a computer system, some of them are discussed in the following paragraphs.

Isolation

An important benefit of virtualization is isolation. The VMM is able to manage all guests enforcing the isolation property. Each guest is allocated into a partition that is protected from the other partitions. A software guest can run on a dedicated virtual machine, without being aware of the presence of other guests. The isolation property enhances the system reliability. In fact, thanks to the isolation, a fault caused by a guest do not affect other partitions hosting other guests of the same machine.

It is worth mentioning that nowadays, instead of pure isolation, virtualization is used in architectures where guests want to communicate. It is then important for a VMM to provide a communication infrastructure to support inter-guest cooperation. This must be done preventing interferences between guests.

Minimized trusted computing base

The *trusted computing base (TCB)* of a system is the set of all hardware and software components that are critical to its security, in the sense that bugs or vulnerabilities occurring inside the TCB might compromise the security properties of the entire system. A given piece of hardware or software is part of the TCB if and only if it is designed to be part of the mechanism that provides security to the system. In operating systems, TCB consists of those functionalities that run under a privileged mode and manage the system resources. Therefore, having a large TCB increases the attack surface of the whole system.

With hardware virtualization, the problem of attacking the TCB is addressed minimizing the TCB itself. In fact, VMM code is the only one running in privileged mode, while the guest code executes with a non-privileged mode. Consequently, the TCB of a virtualized system is only the VMM code. With a minimal TCB, the attack surface of the system is minimal. Besides, since it is is security-critical for the whole system, having a minimal code for the TCB open more chances to a formal verification if compared with an OS kernel.

Security

Virtualization enhances the security of a system. This is thanks to the isolation and the minimized trusting computing base that characterize a virtualized system. In fact, with the isolation property an untrusted guest can execute on the same machine without compromising the other trusted parts of the system. Whereas a minimal TCB reduces the surface attack and allows a formal verification and analysis.

More in general, a VMM satisfying the VMM properties stated by Popek and Goldberg has a complete control over the system resources. This property enables possibilities for additional security services that a VMM can provide to their guests. These services can exploit the isolation property bringing benefit for a guest OS without its modification. The improved security offered by the hardware virtualization is one of the basic motivations leading this thesis work as it will be clear from Chapter 3.

2.2 The Prosper Hypervisor

Today the use of embedded systems in every day life is increasing dramatically. For example, a mobile phone is one of the most common embedded systems used for digital communication. More in general, embedded systems are being used more and more for several purposes and in different domains: entertainment, media and finance, control systems in industry and health. Consequently, given their employment in many areas, security in embedded systems is becoming a relevant research topic [22].

Security issues in embedded systems include different aspects related to reliability (low failure probability, robustness during execution, etc), but also related to protection from malicious software attacks. The second aspect is particularly critical considering that open source software is often used for embedded system. The use of open source software brings several benefits and open possibilities for improvements by a large community. However, source code and documentation availability might introduce some vulnerabilities in embedded system security, giving more chances for malicious attacks. One of the main focus of the Prosper project [3] is on addressing software attacks in embedded systems.

Virtualization can be a valid approach to enhance the security level in the realm

of embedded systems. A Virtual Machine Monitor runs at the most privileged execution level of the system providing isolation and security services to a software guest (see Section 2.1.1). Thanks to this benefits, trusted and untrusted applications can coexist sharing the same hardware without interferences, and relying on the security services provided by the VMM. In addition, with a tamper-resistant, minimal and formally verifiable trusted computing base (TCB), the embedded system security is enhanced. TCB can also provide the infrastructure for a security service such as monitoring or access control.

The Prosper project [3] has the goal of building a framework for fully verified and secure Virtual Machine Monitors for embedded systems, also denoted as *hypervisors*. The main result [11] [20] [10] [12] achieved so far is the design and the implementation of a fully verified tiny hypervisor targeting ARMv7 architecture [27] [29]. The Prosper hypervisor exploits the MMU (Memory Management Unit) security separation to embedded systems security. Thanks to the separation property, the Prosper hypervisor supports parallel execution of multiple para-virtualized guests running in user mode.

2.2.1 ARM architecture

A good overview about ARM architecture can be found in [27] . The article describes in a synthetic and readable way the architecture and suits perfectly with the knowledge requested for this thesis work.

Advanced RISC Machine (ARM) is a family of *instruction set architectures (ISA)* for computer processors based on a *reduced instruction set computer (RISC)* architecture developed by the British company ARM holdings. An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O. An instruction set architecture provides a specification of the machine, and the native commands implemented by a particular processor.

The ARM ISA is a reduced instruction set, that is a particular CPU design strategy based on a small and simple set of instructions. The idea behind a RISC ISA is that simple instructions are highly-optimized, therefore a high level of performance can be achieved. On the other hand with a RISC ISA more responsibility is put on the compiler. *Complex instruction set computer (CISC)* is the alternative

design strategy for CPUs. A CISC ISA consists of a set of complex instructions that rely more on functionalities provided at hardware level.

Today, ARM is the commonest ISA for embedded systems architecture, and this is mainly thanks to a combination of features that makes it particularly suitable for this kind of systems. First, ARM cores are very simple compared to most other general-purpose processors, which means that they can be manufactured using a comparatively small number of transistors. Second, both the ARM ISA and the pipeline design are aimed to minimize energy consumption, that is a critical requirement in mobile embedded systems. Third, the ARM architecture is modular: the only mandatory component of an ARM processor is the integer pipeline; all other components, including caches, MMU, floating point and other co-processors are optional, which gives a lot of flexibility in building system ARM-based CPUs.

The ARM ISA is a load-store architecture, that is, instructions that process data operate directly on registers and are separated from instructions that access memory. All ARM instructions are 32-bit long and most of them have a three-operand encoding. There is also an ARM extension that introduces a subset of 16-bits instructions called *Thumb instructions*. This extension is introduced to achieve a higher code density for embedded applications, and considers a compressed version of the most commonly used 32-bit instructions. The ARM architecture specifies 16 general-purpose registers and provides support for coprocessors allowing the architecture to be extensible in case of specific applications.

Registers

The ARM ISA provides 16 general-purpose registers in the user mode. Register 15 is the program counter, but can be manipulated as a general-purpose register. The general-purpose register number 14 is used as a link register by the branch-and-link instruction, that is provided by the ARM architecture to support conditional execution of arbitrary instructions. Register 13 is typically used as stack pointer, although this is not mandated by the architecture.

The *current program status register (`CPSR`)* contains four boolean condition flags (`Negative, Zero, Carry, and oVerflow`) and four fields containing the execution mode of the processor. The `T` field is used to switch between ARM and Thumb instruction sets. The `I` and `F` flags enable normal and fast interrupts respectively. Finally, the *mode* field selects one of seven execution modes

that include a single non-privileged mode: **user mode** used for programs and applications running on the operating system, and six **privileged modes**:

- *Fast interrupt processing mode*: is enabled if the processor receives an interrupt request from a fast interrupt source.
- *Normal interrupt processing request*: is enabled when the processor receives an interrupt signal from any other interrupt source.
- *Software interrupt mode*: is enabled when the processor encounters a software interrupt instruction. Software interrupts are the standard way to invoke operating system services on ARM.
- *Undefined instruction mode*: is entered if the processor tries to execute an instruction that is not supported by the main integer core or some coprocessors.
- *System mode*: is used when privileged operating system tasks are running.
- *Abort mode*: is entered in correspondence of a memory fault.

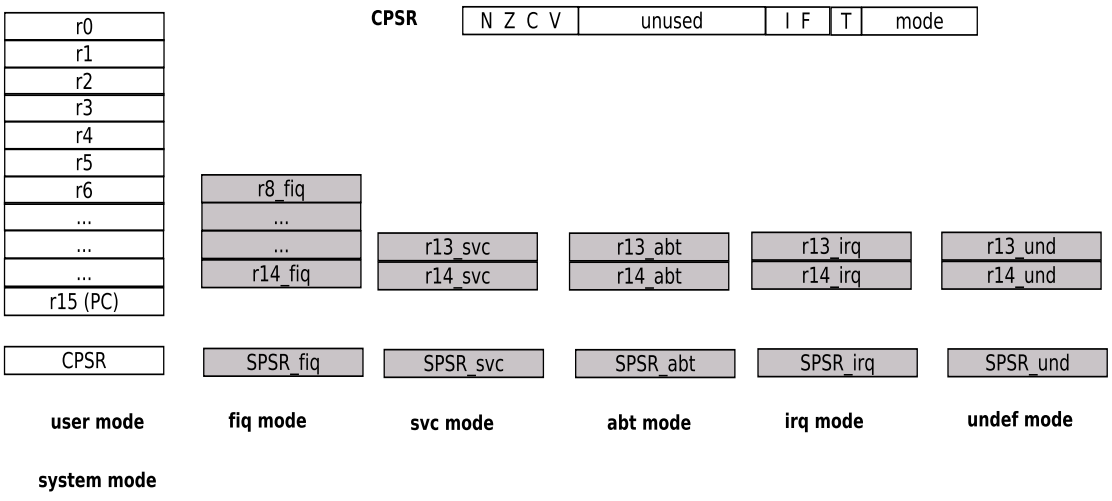

Figure 2.3: ARM registers

Except of the boolean conditions field, `CPSR` is accessed only in a privileged mode. In addition to the 16 registers accessible in user mode,there are several

registers accessible in privileged modes only. For example, each SPSR (Saved Program Status Register) is used to store a copy of the value of the CPSR register when an exception is raised. Privileged modes that are activated in response to exceptions have their own R13 and R14 registers, which allows to avoid saving the corresponding user registers on every exception. For the quick handling of a fast interrupt request (FIQ), ARM provides 5 additional registers available only in the fast interrupt processing mode.

Exceptions

The ARM architecture defines the following types of exceptions (listed in the order of decreasing priority):

- *Reset*: starts the processor from a known state and makes all other pending exceptions irrelevant.
- *Data abort*: is raised by the memory management unit when a load or store instruction violates memory access permissions.
- *Fast interrupt*: is raised whenever the processor receives an interrupt signal from the designated fast interrupt source
- *Normal interrupt*: is raised whenever the processor receives an interrupt signal from any non-fast interrupt source.
- *Prefetch abort*: is raised by the memory management unit when access permissions are violated during an instruction prefetch.
- *Software interrupt*: is raised by a special instruction, typically for an operating system functionality request.
- *Undefined instruction*: is generated when trying to decode an instruction that is not supported by the main integer core nor by one of the coprocessors.

All exceptions, except from the reset exception, are handled in a similar way: the processor switches to the corresponding execution mode presented in the previous section, saves the address of the instruction following the exception entry instruction in R14 of the new mode, saves the old value of CPSR to SPSR of the new mode, disables IRQ (in case of a fast interrupt, FIQ is also disabled), and starts execution from the relevant exception vector.

Coprocessors

A coprocessor is a computer processor used to supplement the functions of the primary processor (the CPU). The ARM architecture supports a mechanism for extending the instruction set with additional coprocessors. For example, the ARM floating point unit is implemented as a coprocessor. Of more interest for this thesis is the system control coprocessor that is for the *Memory Management Unit (MMU)* and *translation lookaside buffer (TLB)*.

Memory Management Unit The Memory Management Unit (MMU) is a computer hardware unit which has the role of controlling all memory references. An MMU is the unit that effectively performs the virtual memory management, handling at the same time memory protection and cache control. General-purpose ARM-based systems are equipped with the MMU that provides a virtual memory model similar to conventional desktop and server processors. Through coprocessor 15 [1] of the ARM architecture, the MMU can be enabled. In case the MMU is disabled, when the CPU accesses memory, the virtual memory address is mapped to the same physical address (no translation is performed). If the MMU is enabled, the virtual address is translated to a physical address and the memory access is checked according to *access permissions*. This is done based on the mechanism of *page tables* that is discussed in the next paragraphs.

In order to improve virtual addresses translation, the MMU uses a cache called *Translation lookaside buffer (TLB)*. TLB contains the recently accessed mappings to speed-up the translations phase.

Page tables The MMU ARM architecture supports a two-level hierarchy for its page table structure, with page sizes of 1MB, 64KB, 4KB, and 1KB. For The Prosper hypervisor, we consider a page size of 1MB that are also called sections. A first-level page table (also called *master* page table) contains 4096 entries allowing up to 4GB of virtual memory. Each entry (also called page table descriptor), can be either unmapped, or it contains a pointer to a second-level table

[1] The coprocessor 15 of an ARM chip is used to configure and control the ARM core modules including the caches and the MMU.

or a base address for a section of 1MB (in this case we have a section entry). A second-level page table has 256 entries each mapping 4KB of contiguous virtual memory. Figure 2.4 shows an overview of the first and second level page table.

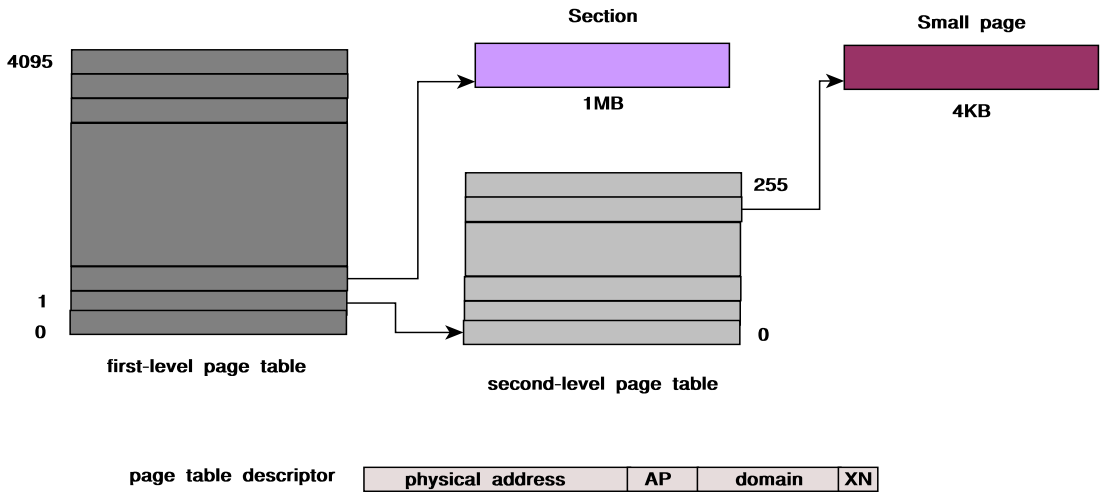

Figure 2.4: 2-level page tables

Each page table descriptor is composed by three fields: the first one contains the physical address of the region where the virtual address is mapped, the second and third fields contain access permission bits and the domain of the mapped region respectively, that are needed for the memory protection mechanism.

When a mapping is needed, the TLB is searched first, if the mapping is not found the MMU performs the translation through a page table walk. That is the current master page table is used for the translation of the virtual address. If the mapping searched is section-mapped, the physical address is returned and the translation is finished. If the mapping is page-mapped (through a second-level page table), an additional level of translation is required. This translation is based on the second-level page table that is pointed by the master table descriptor. In both cases, the retrieved mapping is inserted into the TLB, possibly after the invalidation of an old entry if the cache is full.

Memory protection mechanisms There are basically two mechanisms used by the MMU for memory protection. The first one is based on access permissions. Each page table descriptor contains access permission bits stating whether the mapped physical region can be accessed in readable and/or writable mode. A particular bit called *execute never (XN)* states if the mapped region can be executable. In this thesis we consider that the XN field is an access permission as well. A memory access is allowed only if it satisfies the access permissions retrieved during the translation phase, otherwise an exception is raised. The second protection mechanism consists of a *domain based memory access*. Every virtual memory page or section belongs to one protection domains. The current domain is contained in the *domain access register* of coprocessor 15. A running process can be either a manager of the domain, which means that it can access all pages belonging to this domain bypassing access permissions, a client of the domain, which means that it can access pages belonging to the domain according to their page table access permission bits, or can have no access to the domain at all.

2.2.2 Formal Model of the ARM architecture

In this section we present the formalization of the ARMv7 architecture (described in Section 2.2.1). This formal model is used in this thesis report as a building block to describe the behaviour of a system running on top of the Prosper hypervisor. In particular, this formalization will be used to describe a top level specification of the system behaviour running on top of hypervisor and monitored by the Prosper monitor. The formalization is also used to describe the formal verification performed in this work.

Execution modes

The execution modes of an ARM CPU are formalized by the set *mode*:

$$mode \stackrel{\text{def}}{=\!=} \{usr, svc, abort, undef, irq, fiq, sys\}$$

The non-privileged mode *usr* is used by the application processes, while all other modes are privileged and they are used to execute kernel activities. We

define privileged and non-privileged modes as sub-sets of *mode*:

$$usrmode \stackrel{\text{def}}{=\!=} \{usr\}$$
$$privmode \stackrel{\text{def}}{=\!=} mode \setminus \{usr\}.$$

Registers

The 16 ARM registers are formalized as a function *regs* that takes a register index, an execution mode and returns the register value if it is accessible in the input mode, otherwise it returns an undefined value. We formalize indexes and register values as sets:

$$idx \stackrel{\text{def}}{=\!=} word8$$
$$regval \stackrel{\text{def}}{=\!=} word32 \cup \{\bot\}$$

We denote with *word8* and *word32* an 8-bits and 32-bits values respectively. We denote with $\bot$ the undefined value. Therefore, the function *regs* can be formalized as follows:

$$regs : idx \times mode \longrightarrow regval$$

The Program status register is formalized as a function that takes a privileged mode and returns as output the register content:

$$psrs : privmode \longrightarrow word32.$$

Coprocessors

In this work, the only coprocessor of interest is the system control coprocessor 15 that controls the Memory Management Unit (MMU). We are interested in three registers of 32-bits of the coprocessor 15: the $c1$ represents if the MMU is enabled or not, the $c2$ gives the base physical address of the current master page table, and the register $c3$ identifies the current status of the domains. These registers are coregisters and we represent them as a tuple *coregs*:

$$coregs = \langle c1, c2, c3 \rangle$$

Machine states

We consider a physical memory of 4GB that is addressed by a physical address of 32-bits. We denote a physical address with *phy_addr* and a value of memory with *memval*. These are basically *word32* and *word4* values. The physical memory is represented by a function *mem*:

$$mem : phy\_addr \longrightarrow memval$$

Finally, an ARM machine state σ is formalized as a tuple:

$$\sigma = \langle m, regs, psrs, coregs, mem \rangle \in \Sigma$$

Where: Σ is the set of all machine states, $m \in mode$, $regs, psrs, coregs$ and mem are as described above.

ARM machine behavior

An ARM machine behaviour is modeled by a transition system. The state transition relation is $\rightarrow_l \subseteq \Sigma \times \Sigma$, where $l \in mode$, and a transition is performed by the execution of an ARM instruction. A non-privileged transition $\sigma \rightarrow_l \sigma'$ with $l \in usrmode$ starts and ends in non-privileged states. Namely, if $\sigma = \langle m, regs, psrs, coregs, mem \rangle$ and $\sigma' = \langle m', regs', psrs', coregs', mem' \rangle$, then $m, m' \in usrmode$. A privileged transition $\sigma \rightarrow_l \sigma'$ with $l \in privmode$ involves at least one state in privileged mode. The raising of an exception is modelled by a transition that starts in user mode and enables a privileged level $l \in privmode$.

Memory Management Unit behavior

The main functionality provided by the MMU is to mediate all memory accesses, therefore it is invoked whenever an instruction that requires a memory access is performed. The MMU translates a virtual addresse to a physical address and mediates all memory accesses based on the protection mechanisms discussed in Section 2.2.1.

Given an ARM machine state and a virtual address of 32-bits, the MMU returns the physical address and the retrieved access permissions. These access permissions are represented as a 3-bits value and state whether the computed physical address can be accessed in readable, writable or executable mode. Both

physical address and access permissions are computed using the master page table pointed by the coprocessor register $c2$, and the domain status and the current mode identified by the coprocessor register $c3$. The MMU behavior is formalized by the function *mmu*:

$$mmu : \Sigma \times virt\_addr \longrightarrow phy\_addr \times access\_mode.$$

Where *virt_add* is a *word32* value, and *access_mode* is a *word3* value. For instance, given a virtual address va and a machine state σ, we can apply *mmu* as follows:

$$mmu(\sigma, va) = \langle pa, (ex, w, r) \rangle$$

The application of the *mmu* function returns a pair $\langle pa, (ex, w, r) \rangle$, where pa is the physical address, (ex, w, r) are instead the access permissions: ex is 1 if and only if the address pa can be executed, w is 1 if and only if the address pa can be written and r is 1 if and only if the address pa can be read.

2.2.3 Hypervisor: design and isolation property

The Prosper hypervisor supports the ARMv7 architecture and is able to host a set of trusted guests along with an untrusted Linux guest [10]. The Linux guest may require a dynamic allocation of the memory, whereas all other guests have a static memory configuration. The trusted guests can provide for example some support services to the Linux kernel.

The Prosper hypervisor allows the execution and management of resources by all guests *without interferences* [12] [10]. This is guaranteed by ensuring the isolation property.

The hypervisor provides a virtualization mechanism of the memory subsystem allowing each guest to manage dynamically its own memory hierarchy and to enforce its own memory protection mechanisms.

The Linux kernel hosted by the hypervisor is a paravirtualized OS. Consequently the Linux kernel is aware of the hypervisor mediation. The hypervisor provides to Linux an API for a secure access to the hardware resources. This mediation allows the Prosper hypervisor to reach the requirement of *resources control* discussed in Section 2.1.

One of the most important system resource is the system memory. Therefore, the Prosper hypervisor has a complete control on the memory configuration. As

introduced in Section 2.2.1, the main functionality of the MMU is managing the virtual memory mapping that is configured through a set of page tables residing in physical memory. Since page tables state how physical memory can be accessed by guests, they are a security critical part of the system and must be not directly manipulated by untrusted guests such as the Linux kernel. However, the Linux kernel needs to access dynamically page tables to set the memory layout. Therefore, page tables accesses are mediated by the hypervisor, thus providing an appropriate secure access that is called *MMU virtualization*.

Virtualization mechanism

The virtualization mechanism implemented by the Prosper Hypervisor is based on *direct paging*. This is a common approach for the memory subsystem virtualization adopted for example in the Xen hypervisor [4] [8]. In this approach the page tables are located inside the guest memory simplifying the OS adaptation required by the paravirtualization. In order to guarantee the isolation, the hypervisor makes the page tables read-only to the OS providing an API to manipulate them safely.

Direct paging allows a guest to manipulate directly a page table as long as it is in passive state, that is not used by the MMU (also called active state). Once a page table is activated, all OS accesses are performed invoking the hypervisor through an appropriate API. This invocation is also called *hypercall*.

As presented in Section 2.2.1, the ARM MMU supports a two-level hierarchy for its page table structure. First-level page tables contain 4096 entries each of 32 bits, while second-level page tables contain 256 entries. Therefore, a page table of first-level is of 16KB, and the one of second-level is of 1KB. Physical memory is of 4GB and it is logically fragmented into blocks of 4KB, consequently a first-level page table occupies 4 blocks, while a single physical block can contain up to 4 second-level page tables.

Since page tables are kept in the guest space, this simplifies the OS adaptation. However, this implies that the hypervisor must protect the page tables from accesses in writable mode. In particular, the hypervisor must prevent all modifications of memory configuration that can be performed without the hypervisor mediation. To allow the hypervisor to protect the page tables, the virtualization mechanism types the physical blocks. Each physical block can have one of the following types:

- *data:* the block does not contain sensitive data and can be written by a guest.
- *L1:* the block contains a part of a first-level page table.
- *L2:* the block contains 4 second-level page tables.

Typing each physical block allows the hypervisor to write-protect active page tables enforcing *page type constraints*. These constraints require that a guest might write only physical blocks of type data. In order to fulfil this constraint, the virtualization mechanism maintains *reference counters* that track:

- For each block typed L2 the number of L1 page tables entries that point to one of the 4 L2 page tables residing in the physical block.
- For each block of type data, the number of entries (L1 or L2) that point to the physical block and are writable in user mode.

A guest can change the type of a physical block (for example by allocating or freeing a page table) only if the corresponding reference counter is zero. The hypervisor enforces this policy in addition to the general isolation policies (e.g. a guest may only modify a page within its assigned physical memory region) whenever the MMU configuration is updated. Since all MMU updates are performed through a dedicated API, the hypervisor performs the necessary checks to enforce security policies each time a hypercall is executed. The API offered by the hypervisor includes handlers for the manipulation of the MMU configuration, namely for:

- Page tables creation.
- Page tables freeing.
- Mapping of a page table entry.
- Unmapping of a page table entry.
- Setting a master page table as the current page table.

Top Level Specification

The behaviour of the hypervisor is defined as a transition system. This specification, called the Top Level Specification (TLS), models the behaviour of a system in which an arbitrary guest is running on top of an ARMv7 CPU with MMU support, in alternation with executions of abstract handler events. Handler events are performed in correspondence to ARMv7 privileged instructions, in response to a hypercall. These events imply invocations of the hypervisor handlers as atomic transformations H_a operating on an abstract machine state. Abstract states are concrete ARMv7 states extended by auxiliary data structures such as page types or reference counters that reflect the internal state of the hypervisor. Auxiliary data structures are called *abstract hypervisor state*.

Formally, the TLS state is represented as a tuple $\langle \sigma, \eta \rangle$, consisting of an ARM state $\sigma \in \Sigma$ and an abstract hypervisor state $\eta = \langle pgtype, pgrefs \rangle$. More precisely,*pgtype* is a function for memory block typing and *pgrefs* is a function tracking reference counters of memory blocks.

$$pgtype : [0, 2^{20}) \longrightarrow \{D, L1, L2\}$$

For each possible physical block of 4KB, *pgtype* returns the block type that can be data (D), first-level page table ($L1$), or second-level page table ($L2$).

$$pgrefs : [0, 2^{20}) \longrightarrow \mathbb{N}.$$

For each possible physical block of 4KB, $pgrefs$ returns the number of references to it in the memory.

The TLS interleaves standard non-privileged transitions with abstract handler invocations. The TLS transition relation $\langle \sigma, \eta \rangle \rightarrow_{i \in \{0,1\}} \langle \sigma', \eta' \rangle$ is defined as the following:

$$\frac{\sigma \xrightarrow{op}_l \sigma' \qquad l \in usrmode}{\langle \sigma, \eta \rangle \xrightarrow{op}_0 \langle \sigma', \eta \rangle}\ usr \qquad \frac{\sigma \xrightarrow{op}_l \sigma' \qquad l \in privmode}{\langle \sigma, \eta \rangle \xrightarrow{op}_1 H_a(\langle \sigma', \eta \rangle)}\ priv$$

The first inference rule *usr* states that an instruction *op* executed in non-privileged mode that does not raise exceptions behave equivalently to the standard ARMv7 semantics and does not affect the abstract hypervisor state. The second inference rule *priv* states that whenever an exception is raised, the hypervisor is invoked through a hypercall, and the reached state is resulting from the execution of the handler H_a.

Isolation Property

Guaranteeing spatial isolation means confining each guest to manage only the partition of memory assigned to its uses. All partitions assigned to guests are statically defined. In general, no security property can be ensured if the starting state of the TLS is inconsistent with the security property itself. Therefore, a system invariant is defined $I\langle\sigma,\eta\rangle$ to constrain the set of consistent initial states of TLS. In addition, a set Q_I of all the TLS states satisfying the system invariant is introduced.

The system invariant $I\langle\sigma,\eta\rangle$, consists of two predicates: RC and TC. RC ensures soundness of the reference counters tracked by $pgrefs$, and TC guarantees that the state σ is well typed according to *pgtype*. The reference counter is sound ($RC(\langle\sigma,\eta\rangle)$), if for every physical block b, the reference counter $pgrefs(b)$ is equal to $\sum_{i\in\{0\ldots 2^{20}\}} count(\sigma,\eta,b,i)$, where *count* is a function that counts the number of references to the block b, according to the reference counter policy. A system state is sound ($TC(\langle\sigma,\eta\rangle)$), if the MMU is enabled, the current master L1 page table is inside a physical block of type L1 and each physical block b of type different from data ($pgtybe(b) \neq D$), contains a sound page table ($sound(\sigma,\eta,b)$). The predicate *sound* ensures, among other things that: (1) the page table grants writable access only to physical blocks of type data, (2) the page table forbids any writable access in user mode to blocks outside the guest partition and (3) each page table entry using a second-level page table points to a physical block typed L2 (See [12] for more details).

The security properties that are ensured by the Prosper hypervisor are stated by three theorems.

Theorem 1. *Let* $\langle\sigma,\eta\rangle \in Q_I$ *and* $i \in \{0,1\}$. *If* $\langle\sigma,\eta\rangle \rightarrow_i \langle\sigma',\eta'\rangle$ *then* $\langle\sigma',\eta'\rangle \in Q_I$.

Theorem 1 states that, starting from a consistent state, an arbitrary TLS transition ends in a consistent state.

The hypervisor guarantees also data separation properties. In order to introduce theorems regarding these properties, some concepts related to *state observations* are needed. The observation of a guest in a state ($\langle\sigma,\eta\rangle$) is represented by the structure $O_g(\langle\sigma,\eta\rangle) = \langle uregs, cpsr, mem_g, coregs\rangle$ of user registers $uregsg$, control register $cpsr$, guest memory mem_g and coprocessor registers $coregs$. The

register *cpsr* and the coprocessor registers are visible to the guest since they directly affect the guest behaviour, and do not contain any information the guest should not be allowed to see, even if all writes to the coprocessor registers must be mediated by the hypervisor. The remaining part of the state (i.e. the content of the memory locations that are not part of the guest memory or the special registers) and, again, the coprocessor registers constitute the secure observations $O_s(\langle\sigma, \eta\rangle)$ of the state, which guest transitions are not supposed to affect.

Data separation properties are expressed through the following two theorems:

Theorem 2. *Let* $\langle\sigma, \eta\rangle \in Q_I$*, If* $\langle\sigma, \eta\rangle \rightarrow_0 \langle\sigma', \eta'\rangle$*, then* $O_s(\langle\sigma, \eta\rangle) = O_s(\langle\sigma', \eta'\rangle)$

Theorem 2 states the *non-exfiltration* property. This guarantees that a transition executed by the guest does not modify the secure resources.

Theorem 3. *Let* $\langle\sigma_1, \eta_1\rangle, \langle\sigma_2, \eta_2\rangle \in Q_I$ *and assume that* $O_g(\langle\sigma_1, \eta_1\rangle) = O_g(\langle\sigma_2, \eta_2\rangle)$*. If* $\langle\sigma_1, \eta_1\rangle \rightarrow_0 \langle\sigma_1', \eta_1'\rangle$ *and* $\langle\sigma_2, \eta_2\rangle \rightarrow_0 \langle\sigma_2', \eta_2'\rangle$*, then* $O_g(\langle\sigma_1', \eta_1'\rangle) = O_g(\langle\sigma_2', \eta_2'\rangle)$

Theorem 3 states the *non-infiltration* property that is a non-interference property. Namely, a transition executed by the guest depends only on its observations.

All theorems 1, 2 and 3 are satisfied by the Prosper hypervisor. This is formally proved in HOL4 theorem prover, based on the hypervisor TLS definition in the HOL4, and on the HOL4 model of the ARM architecture that is developed at Cambridge [16].

2.3 Automated Theorem Proving

The goal of Automated Theorem Proving (ATP) is to prove automatically that a given statement expressed as a theorem follows logically from a set of hypothesis expressed as axioms and inference rules. The ATP deals with the development of computer programs that automatizes the process of proving that the theorem is a logical consequence of the hypothesis.

ATP can be applied to several domains, for example in mathematics it helps in proving a conjecture given a set of assumptions. In computer science ATP is used in artificial intelligence as a tool for the automated reasoning. ATP is also a formal method for the software and hardware verification. In the area of formal verification, ATP deals with the development of formal models that

specify mathematically a given system, and the proof of correctness of these models with respect to a specified requirement or property. The mathematical model is expressed as *formulae* of a logic, and the property to verify is the theorem to be proved as a logical consequence of these formulae.

There are many theorem provers that support the process of proving automatically a theorem. A theorem prover provides a logical system with a set of axioms, theorems and inference rules that supports a user in writing a theorem's proof. The logic provided by a theorem prover can be for example a simple propositional logic, a temporal logic, a first-order logic or a higher-order logic.

A fundamental part of this work consists of the formal verification of a security module that is run as a guest of the Prosper hypervisor. The formal verification is assisted by the HOL4 theorem prover [17] that supports a higher-order logic.

2.3.1 The HOL4 thorem prover

The HOL4 is an ML-based environment supporting both specification and formal proofs in *higher order logic* [17]. The interactive environment of HOL4 allows to build new theories or to simply write personalized procedure of verification, using an ensemble of automated reasoning tools that are freely available. In the following paragraphs we introduce a few concepts related to the HOL4 logic, and we briefly discuss how a proof can be supported by the HOL4 theorem prover, the reader if interested can see [17] for more details.

The HOL4 Logic

The HOL4 system supports a higher order logic, that is a version of predicate calculus extended by two additional aspects:

- Logical variables can also range over functions and predicate.
- The logic is typed

The HOL4 Logic is essentially Church's Simple Type Theory [30]. It is based on simple types *ty*, that are used to build typed-lambda calculus terms *tm*. The HOL4 logic has a set-theoretic semantics. Types denote sets and terms denote members of these sets.

There are four kinds of types in the HOL logic that can be described by the following grammar:

$$ty := \alpha \mid c \mid (ty_1, ..., ty_n)op_n \mid ty_1 \rightarrow ty_2$$

Where α denotes type variable, c denotes an atomic type (for example the standard atomic type *bool*). A compound type is denoted with $(ty_1, ..., ty_n)op_n$ where $(ty_1, ..., ty_n)$ are the argument types and op_n is a type operator of arity n (for example *prod* is a type operator of arity two which denotes the cartesian product operation). The function type $ty_1 \rightarrow ty_2$ denotes the set of all functions from the domain set of type $ty1$ to the codomain set of type ty_2.

A term of the HOL4 logic is an expression that denotes an element of the set denoted by the type associated to the term. There are four types of terms in the HOL4 logic that can be described by the following grammar:

$$tm := x \mid c \mid tm1\ tm2 \mid \lambda x.tm$$

Where x denotes a variable term, c denotes a constant term, $\lambda x.tm$ denotes a λ-abstraction term, and $tm1\ tm2$ denotes the function application term.

The interface to the HOL4 logic is the functional programming language ML. Terms of the HOL4 logic are represented in ML by an *abstract data type* called `term`, and they are introduced between double back-quote marks. The ML parser for HOL4's terms include an inference type checker that is able to assign a type to each well-formed term. A term is well-formed if it can be derived according to the logic syntax.

The logic of HOL4 is implemented by a small kernel, that includes the initial signature, the primitive rules of inference, the axioms and the primitive definition principles of the logic. These principles allow the extension of the logic in a consistency-preserving way.

Proving with HOL4

A formal proof can be seen as a sequence of steps, each one consists of the application of an axiom or an inference rule. The last element of the proof is the theorem. In HOL4 theorems are represented as values of an abstract type *thm*. The only way to create a theorem is by providing a proof. This follows from the application of ML functions that represent inference rules to axioms or to previously computed theorems.

There are two methodologies that are supported by HOL4 for performing an interactive proof: the forward methodology and the goal oriented methodology.

The forward methodology Based on the forward methodology, the proof starts from the appropriate axioms, and through the application of the right inference rules, the theorem to be proved is reached as result at the end of the proof. A forward proof requires the selection of the axioms and the inference rules to be used before starting the interactive proof. Also the order of the application of axioms and rules must be decided. The knowledge requirement of all the proof steps in advance makes the forward methodology not easy to conduct in many cases.

The goal oriented methodology In the goal oriented methodology, a proof is called backward proof and it is the the reverse of the forward proof. This methodology is based on the concept of *tactic*, proposed by Robin Milner in 1970. A tactic is a function that splits a *goal* into *subgoals*, recording at the same time the reason that makes the goal solved once all subgoals are proved. A goal oriented proof starts with the main theorem to be proved, and proceeds by specifying a tactic that splits the main goal into subgolas. The intuition is that a subgoal can be easier to solve compared with the main goal, for example by matching an axiom, or with an automated reasoner provided by the HOL4 system.

A goal in HOL4 is a pair $([t_1, ..., t_n], t)$ where the first component contains a term list representing the assumptions, whereas the second component contains the term goal. The achievement of solving the goal results in a theorem. A tactic is an ML function that applied to a goal generates subgoals and a *validation* that is a justification function of why solving the subgoals is a solution of the goal represented by t. The validation is basically an ML function representing the inference rule that allows to derive the main goal from the subgoals.

The HOL4 theorem prover provides many automatic reasoners and proof assistants that support the user during an interactive proof. These are organized as tactics or as procedures. The user can also write personalized procedures and tactic for its own proofs.

The HOL4 theories

A HOL4 theory is a collection of valid HOL4 term definition, axioms and theorems. A theory can be loaded during an interactive session and used in a proof. Several mathematical concepts and models are defined and provided to users as theories.

The Prosper hypervisor model is provided as a theory `hypervisor_modelTheory`.

Chapter 3

Runtime monitoring for Prosper

This chapter presents the main contribution of the thesis, namely the definition of a validation mechanism for a security property guaranteeing protection against code injection attacks, and the proof of correctness of the validation mechanism based on assumptions on the underlying architecture, that exploits the Prosper hypervisor and the isolation property that it guarantees in a virtualization context.

The next section describes more in details the motivations already sketched in the Introduction, and provides an outline of the rest of the chapter.

3.1 Motivations and Goals

Runtime monitoring is a computing system analysis technique based on the observation of a running system and the consequent reaction in case of violation of certain security properties. The main goal of a runtime monitor is to ensure a security property preventing incorrect system states. Runtime monitoring is employed whenever a static analysis of the system behaviour against a security property is infeasible.

One of the most common applications of runtime monitoring, that is also in the focus of this thesis, is on Operating Systems. An OS is responsible for managing the computer system resources (files, memory, devices, etc). This management includes the allocation of a resource to the different user applications and its pro-

tection from unauthorized accesses through an appropriate access control policy.

A common approach is to realize the access control policy with a runtime monitor that controls all the accesses to the resource. This is achieved deploying the runtime monitor as a kernel security module. The kernel is usually explicitly developed to support this security module. In fact, whenever a critical operation is going to be performed by the kernel, the security module is explicitly informed. The Linux Security Module (LSM) [33] is an example of a runtime monitor for the Linux kernel that supports and provides access control security policies. The basic abstraction of the LSM interface is to mediate all the accesses to the internal kernel objects. The LSM is invoked each time the Linux kernel would have access to a system resource.

Vulnerabilities in LSM approach

The standard approach of deploying a runtime monitor to protect a system resource as a kernel component has some vulnerabilities. This is mainly due to the kernel's complexity. In fact, the complexity of the kernels makes them susceptible to attacks [25] [9] that can bypass the security policy of the monitor. The design of the LSM allows the security module to mediate the accesses to a kernel object by placing *hooks* at every point of the kernel code where a user system call is about to result in an access to a kernel object. The hook is basically a call to a function provided by the module that implements the protection mechanisms. A malicious program, for example a rootkit that enables privileged access, can disable the mediation of the LSM or even modify the function invoked by a hook.

Provably secure runtime monitor

In general, having a formal proof of correctness of a security critical component enhances the security level of the whole computing system. An OS runtime monitor that protects a system resource from malicious access is certainly a security critical part of the system. Therefore, it is desirable to provide a tamper-resistant and trustworthy runtime monitor. The trustworthiness can be significantly improved if a formal verification of the monitor's security policy against the desired security property can be performed. The tamper-resistance is possible only if the runtime monitor is itself protected from malicious manipulations.

The approach of including a runtime monitor as a kernel security module introduces difficulties in achieving both the trustworthiness and the tamper-resistance of a security module. In fact, it's well known that the OS size and complexity make a formal proof of a kernel module correctness very hard. Also, the OS vulnerability against malicious intrusions prevents from providing a complete tamper-resistant kernel module.

Exploiting virtualization

One of the advantages of the virtualization technology discussed in Section 2.1.3 is the *isolation*. This property allows several software guests to execute in separated and protected partitions without interferences. This property can be strongly related to the tamper-resistance requirement of a runtime security module. For example, a security module running as a guest of a Virtual Machine Monitor can benefit from the isolation property provided by the VMM. Moreover, a direct interaction between the VMM and the security module can allow the runtime monitoring of the guests without affecting the security module integrity.

An additional advantage of the virtualization consists of a *minimal trusted computing base (TCB)* of the virtualized system. This property is related to the trustworthiness requirement of a runtime security module (that is part of the TCB). In fact a minimal trusted computing base is more suitable for a formal verification.

Goals

Given the motivations discussed above, a valuable idea for the Prosper project [3] is to introduce a security module that monitors the system guest of the hypervisor and that takes advantage from the virtualization benefits ensured by the hypervisor. In the following, we will refer to this security module as *the Prosper (runtime) monitor* or simply as *the monitor*.

The first goal consists of giving the general design choices for the Prosper runtime monitor in order to meet the motivations stated above (the design choices are discussed in Section 3.2.1). These choices take into account the infrastructure provided by the Propser hypervisor (the hypervisor is presented in Section 2.2.3). In particular, a desirable feature of the Prosper runtime monitor is to take advantage from the isolation property and the virtualization mechanism provided by the

hypervisor. The second goal consists in analysing, in general, what kind of security properties can be provided by the Prosper runtime monitor. The third goal is the identification of an application for the Prosper runtime monitor. This is done through the identification of a security property, and an appropriate policy, to be enforced in order to protect the Linux kernel guest from code injection attacks (the second and the third goals are treated by Sections 3.2.2 and 3.2.3). The fourth goal consists in providing a formal proof of correctness for the Prosper runtime monitor. In particular, we would like to formally prove that the policy identified is able to ensure the desired security property. The formal proof is presented in Section 3.3 and it has been verified with the support of the HOL4 proof assistant as reported in Chapter 4.

3.2 The Prosper runtime monitor

The Prosper hypervisor [3] [11] [20] [12] [14] is a Virtual Machine Monitor and it satisfies the three properties of a VMM discussed in 2.1.1. In particular, the hypervisor has the complete control of the system resources. Therefore, a monitor that needs to enforce a security property over a system resource can be designed as a hypervisor module. This approach has the advantage of simplifying both the detection of the security critical operations and the monitor mediation on these operations. However, having an additional module of the hypervisor has the inconvenience of increasing the complexity and size of the hypervisor. This violates the requirement of keeping a minimal VMM and a minimized trusted computing base (TCB). Therefore, the approach of deploying the runtime monitor as a hypervisor module suffers from some vulnerabilities that are detected in the LSM approach.

One possible alternative approach is to deploy *the runtime monitor module as a guest* of the Prosper hypervisor. Using a dedicated module on top of the hypervisor permits to decouple the enforcement of the security properties from the other hypervisor functionalities. This choice has mainly two benefits: first, it is possible to maintain a minimal trusted computing base. Second, having the security policy wrapped inside a guest enhances both the tamper-resistance and the trustworthiness of the monitor. In fact, if the monitor runs as a guest, it can take advantage from the isolation properties provided by the hypervisor. This choice

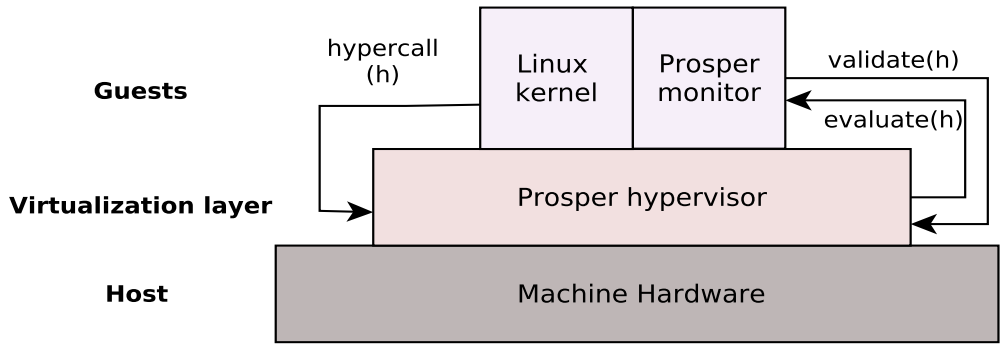

Figure 3.1: System setting

permits to avoid malicious interferences coming from the other guests (for example from a process of the OS running on a different partition of the same machine). In addition, decoupling the runtime security policy from the other functionalities of the hypervisor makes the formal specification and verification of the monitor more affordable.

In this thesis, we consider the system setting shown in Figure 3.1. The system considered has three components:

- The Prosper hypervisor: a virtual machine monitor targeting the ARMv7 architecture [27].
- The Linux kernel: a guest Operating System running in a separated partition. The Linux partition is considered untrusted.
- The Prosper monitor: a runtime monitor running as a guest of the hypervisor and providing a security service for the system.

In the next section we will discuss the functional design choices for the Prosper monitor.

3.2.1 Design choices

In the literature on Operating System security, a runtime monitor enforcing a security policy over a system resource is referred to as *reference monitor* [19] [15] [28]. We rely on this known concept to design the Prosper monitor.

"A reference monitor concept defines a set of design requirements on a *reference validation mechanism*. This mechanism is a functionality provided by the reference monitor. The validation mechanism enforces a security policy over subjects' (i.e. processes, users, etc) ability to perform access operations (i.e. read, write and execute) on system resources (i.e. files, sockets, etc)" [19].

Definition 1 (**validation mechanism**).
A (reference) validation mechanism is the reference monitor functionality that enforces a security property over the system resources.

The Prosper monitor belongs the class of the *Execution Monitoring (EM)* [28] since it is a reference monitor. In [28] it is stated that the monitors of this class are able to ensure a security policy for the *safety properties*. This is a class of security properties that are known and well studied by the literature of formal verification (see [5] [6] [21] for more information about safety properties). In particular the *non-enforceable security policies* principle discussed in [28] states a necessary (and not sufficient) condition that a property P must satisfy in order to be enforced by an Execution Monitor. We instantiate this principle to the Prosper runtime monitor with the following assumption:

Assumption 1 (**non-enforceable security properties for Prosper monitor**).
If P is not a safety property, then a validation mechanism for P from the Prosper monitor does not exist

The Assumption 1 will be taken into account, when the application of the Prosper monitor to the code injection attacks is discussed.

The design requirements established by a reference monitor for a validation mechanism, as discussed in [15] and [19] are:

- *Complete mediation:* the validation mechanism must always be invoked on the critical operations.

- *Tamper-resistance:* the validation mechanism must be tamper-resistant. That is protected from malicious manipulations and cannot be sabotaged or altered.
- *Verifiability:* the validation mechanism must be small enough to allow formal verification and analysis.

The *complete mediation* requirement states that security-sensitive operations must be mediated by the reference monitor. In fact, in order to maintain a correct system state regarding a security property, these operations must be evaluated through the validation mechanism in order to check whether they are safe or not. Intuitively, an operation is safe if the system state resulting after its execution satisfies the security property of interest. We express the complete mediation requirement as a necessary condition to ensure a security property with Assumption 2.

Assumption 2 (**complete mediation requirement**)**.**
A reference monitor is able to ensure a security property P only if it is able to mediate and validate all security-sensitive operations for P.

The tamper-resistance requirement specifies that the validation mechanism cannot be modified by untrusted parts of the system. In fact, a malicious manipulation of a reference monitor functionality can result in a modification of the security policy allowing unsafe operations to be executed. the tamper-resistance requirement gives a second necessary condition to a reference monitor ability of enforcing a security property. This is expressed with Assumption 3.

Assumption 3 (**tamper-resistance requirement**)**.**
A reference monitor is able to ensure a security property P only if it is tamper-resistant.

The verifiability requirement states that the validation mechanism must be as minimal as possible to allow a formal verification. The ability of formally verify if the validation mechanism effectively ensures the desired security policy is fundamental when it comes to ensuring a security property, and this is achievable only if the complexity and the size of the validation mechanism are reduced. Besides, the validation mechanism is a security critical part of the system, therefore it must be

part of the trusted computing base. As discussed in the previous paragraphs, having a minimal trusted computing base is a desirable feature. Therefore, requiring a small validation mechanism is in agreement with maintaining a minimal trusted computing base.

The idea of defining the above design requirements is that, a reference monitor is able to guarantee a security property over all the system states only if it provides a validation mechanism that satisfies the requirements. We summarize this idea through Fact 1.

Fact 1 (**validation mechanism requirements**)**.**
A reference monitor is able to ensure a security property over all the system states only if the validation mechanism satisfies the requirements of complete mediation and tamper-resistance.

The Prosper monitor and the tamper-resistance requirement

Given the system setting considered and shown in Figure 3.1, in order to have a tamper-resistant validation mechanism, the Linux kernel and all the user processes must not be able to manipulate the functionalities provided by the monitor. Thanks to the isolation properties provided by the hypervisor, we assume that the tamper-resistance requirement can be achieved and this is expressed through Fact 2. In fact, the validation mechanism is a functionality provided by the monitor that is a hypervisor guest. Therefore, the validation mechanism is prevented from all malicious interferences and manipulations of the untrusted parts of the system.

Fact 2 (**Prosper monitor and the tamper-resistance**)**.**
The Prosper monitor satisfies the tamper-resistance requirement thanks to the isolation properties ensured by the Prosper hypervisor.

The Prosper monitor and the complete mediation requirement

In order to satisfy the complete mediation requirement, the Prosper monitor must be able to intercept all security-sensitive operations and to invoke the validation mechanism. A security-sensitive operation is evaluated based on the validation mechanism. The outcome of this evaluation states whether the operation can be safely performed or not.

A reference monitor has the goal of enforcing a security property over a system resource (memory, devices, etc). The complete system resources control is of the hypervisor responsibility, according to the VMM properties discussed in Section 2.1.1. This control is possible since a guest is prevented from changing a resource configuration without the hypervisor mediation and control. Therefore, one possibility is to include the monitor mediation each time the hypervisor mediation is requested. This choice allows the enforcement of the monitor security policy over a system resource.

The hypervisor's control over the system resources is achieved providing guests with an API accessible trough hypercalls. It is then reasonable to require that the Prosper monitor must be informed about all the hypercalls that are performed by a guest. For this purpose we identify an interaction protocol between the Prosper hypervisor and the Prosper monitor that is shown in Figure 3.2.

1. For each hypercall invoked by a guest, the hypervisor forwards the hypercall's request to the monitor.
2. The monitor validates the request based on its validation mechanism.
3. The monitor reports to the hypervisor the result of the hypercall validation.

Figure 3.2: The interaction protocol between the Prosper hypervisor and the monitor

The Prosper monitor and the verifiability requirement

The design choice of decoupling the Prosper monitor security service from all the other functionalities of the hypervisor is in agreement with the verifiability requirement. In fact, the validation mechanism of the monitor is separated from the hypervisor services. Therefore, a formal verification of the monitor can be performed independently taking into exam the only validation mechanism. In addition, to meet the verifiability requirement, the checks that need to be performed over a hypercall must be formally identified and be as small as possible.

In order to allow the formal verification of the validation mechanism, this must be specified with a formal model. A desirable requirement for this model is to be

defined at a low level of abstraction. In fact, we would like to have a specification that is as close as possible to a real implementation since all the formal verification is performed over the validation mechanism's model. This requirement makes the work of the formal proof more challenging. In fact, it is not difficult to imagine that a detailed model (i.e defined at low level of abstraction) requires more effort to be formally verified, if compared with a more abstract and general model.

3.2.2 Security properties

In this section we report on the analysis and reasoning done to identify what kind of security properties can be ensured in general by a runtime monitor in the setting shown in Figure 3.1. Then, we discuss an application of the Prosper monitor to protect the Linux kernel guest from code injection attacks through the enforcement of an appropriate security property.

Based on the interaction protocol reported in Figure 3.2, the monitor is invoked each time a guest raises a hypercall. From Section 2.2.3, we know that the API provided by the hypervisor permits a guest to modify the memory configuration. This configuration is managed by the MMU through page tables, as discussed in Section 2.2.1. Consequently, in order to allow a guest to modify its memory configuration, the hypervisor provides a set of hypercalls for creating a new page table, freeing an existing page table, mapping and unmapping of a page table entry and setting a master page table as the current one in the MMU. The Prosper monitor is then able to intercept all the memory layout modifications that are introduced through a page table modification, and this is thanks to the protocol presented in Figure 3.2. Therefore, at this level of abstraction, the kind of security properties that can be enforced by the monitor are those *properties involving some constraints over the memory layout.* We formalize this with Fact 3 taking into account the Assumption 1:

Fact 3 (**enforceable properties for Prosper monitor**)**.**
The Prosper monitor is able to ensure a security property P, with a suitable validation mechanism, only if P constrains the system memory layout and P is a safety property.

Application to code injection attacks

Intercepting all the mappings of the page table entries allows to gain knowledge about whether a physical block is readable, writable or executable. In fact, mapping an entry requires to specify all the information needed to form a page table descriptor (a page table descriptor is shown in Figure 2.4). This information includes the access permissions. Therefore, the monitor can track all the physical blocks in memory that are executable. This can allow the monitoring of the executable code running in the system and eventually enforcing some constraints over the memory layout to ensure some property over the running code. This observation motivates us to consider the application of the Prosper monitor to protect the guest system from code injection attacks.

Code injection is a common attack in computer systems that exploits a software vulnerability, for example a buffer overflow, to inject malicious executable code. This attack, known as *code injection*, is normally performed by a malicious software that is able to write code into a data storage area of another process, and then to run its own code from within this section. In order to run the injected code, this must become part of the executable code of the system. We observe that this event can be intercepted by the Prosper monitor since it is able to track all the executable blocks of the memory.

Informally speaking, we can identify the *executable code of the system* with all the physical blocks that are accessible in executable mode. Therefore, a malicious code injection can be seen as an event that "invalidates" the system code safety. To be able to refer to code safety, we must distinguish between safe and unsafe code. For this reason we rely on the concept of *code signing*.

Code signing is the process of signing a code or script to confirm the software author and guarantee that the code has not been altered or corrupted since it was signed, by use of a cryptographic hash. Many existing code signing systems (for instance Microsoft Authenticode [2]) rely on a public key infrastructure (PKI) to provide both code authenticity and integrity.

In this work we take inspiration from the concept of code signing to define the system code safety. We consider that an executable physical block is safe only if it has a valid signature. All valid signatures are known by the runtime monitor. We refer to this information as *golden image (GI)* and it is held by the monitor. Assuming that the golden image contains only signatures of trusted code,

an injected code can be detected since its signature is not part of the golden image.

More formally, we can identify a golden image GI as a set of signatures, where a signature (for us) can be simply the result of a hash function computed over a physical block content. We introduce more formally the concept of a physical block and a physical block content with Definition 2. Definition 3 instead introduces the concept of a golden image. Notice that all the following definitions use the formal model of the ARM architecture that is presented in Section 2.2.2, where the MMU behaviour and an ARM machine state are described.

Definition 2 (**Physical block, content**).
Given an ARM machine state $\sigma = \langle m, regs, psrs, coregs, mem \rangle$*, each physical block of* σ *is referenced by an identifier* $pb \in [0, 2^{20})$ [1]*. Also, we assume the existence of a function:*

$$content : [0, 2^{20}) \times \Sigma \longrightarrow word4KB$$

that returns the content of a physical block in a system state as a value of $word4KB$*.*

Definition 3 (**Golden Image, valid**).
A golden image GI is a set of signatures that are computed over physical blocks

$$GI = \{s_1, ..., s_n\}$$

Also, we assume the existence of a predicate valid such that, given a machine state σ *and a physical block reference pb:*

$$valid(pb, \sigma, GI) \Leftrightarrow signature(content(pb, \sigma)) \in GI$$

Where signature can be seen as a hash function computed over a value of $word4KB$*.*

We identify the system code given a machine state with the concept of *working set (WS)* that is defined by Definition 4. In particular, the working set contains all those physical blocks that can be accessed in executable mode. Namely, a physical block belongs to the working set, if and only if there exists a virtual address that is translated by the MMU into a physical address of the physical block with executable access permission (see Section 2.2.2, where the MMU behaviour and an ARM machine state are described).

[1] Notice that we will use the notation $pa \in pb$ to denote that the physical address pa is inside the physical block referenced by pb

Definition 4 (**Working set**).
Given an ARM machine state $\sigma = \langle m, regs, psrs, coregs, mem \rangle$ *the working set is formally defined as:*

$$\begin{aligned} WS(\sigma) = &\{\langle pb, content(pb, \sigma)\rangle \mid \exists va, pa, ex, w, r. \\ &\langle pa, (ex, w, r)\rangle = mmu(\sigma, va) \wedge ex \wedge pa \in pb\} \end{aligned}$$

Where pb and content are as defined in Definition 2.

With Defintion 5 we formalize the concept of a *machine state safety* given a golden image, stating that a system state is safe if and only if all the physical blocks in the working set are valid according to the monitor golden image.

Definition 5 (**System safety**).
Given an ARM machine state $\sigma = \langle m, regs, psrs, coregs, mem \rangle$ *and a golden image* GI*:*

$$safe(\sigma) \equiv \forall \langle pb, c\rangle . \langle pb, c\rangle \in WS(\sigma) \Rightarrow valid(pb, \sigma, GI)$$

In order to formally define the security property to be ensured by the monitor, we must introduce a top level specification that characterizes the computations of the system shown in Figure 3.1. This is done only in Section 3.3.1 when we need to present the formal proof. However in the meanwhile we give an intuition for this property with Definition 6 to make the approach of identifying a security policy and a validation mechanism more structured.

Definition 6 (**Property SAFE**).
The (top level specification of the) system depicted in Figure 3.1 satisfies the property SAFE if and only if, given a machine state σ *such that safe(*σ*) holds, if* σ' *is a reachable state from* σ *then* $safe(\sigma')$ *holds.*

The security property *SAFE* belongs to the class of safety properties [5]. Moreover, *SAFE* constrains the system memory layout since *safe*(σ) states a condition over the working set of the state σ. Therefore, the property *SAFE* identified satisfies the necessary condition of *enforceable properties for Prosper monitor* stated by Fact 3.

3.2.3 Security policy and validation mechanism

In this section we report all the reasoning performed to identify a security policy and a validation mechanism needed to ensure the property *SAFE*.

Based on Assumption 2, the Prosper monitor is able to guarantee the property *SAFE* only if it intercepts and validates all the security-sensitive operations for *SAFE*. We identify the security-sensitive operations for *SAFE* as those that change the system working set. Definition 7 defines a sensitive operation for the property *SAFE*.

Definition 7 (**Security-sensitive operations for SAFE**).
Let op be an operation. Then, op is security-sensitive for SAFE if and only if there exist system states σ and σ' such that σ' is reachable from σ by the operation op and $WS(\sigma) \neq WS(\sigma')$.

A simple security policy consists of detecting all the security-sensitive operations of the requests received by the monitor from the hypervisor, and of invoking the validation mechanism over these operations. The request is accepted if and only if it is considered "secure" by the validation mechanism, otherwise it is rejected. The correctness of this policy depends on whether the Prosper monitor is able to intercept all security-sensitive operations for *SAFE*. From the protocol in figure 3.2, we know that the Prosper monitor intermediation is guaranteed only when a hypercall is raised by a guest of the hypervisor. Therefore we can state that:

Proposition 1 (**complete mediation for SAFE**).
The Prosper monitor is able to ensure the property SAFE only if a hypercall is raised for each security-sensitive operation for SAFE.

Proof. Let *op* be a security-sensitive operation for *SAFE* and suppose that *op* is performed without invoking any hypercall. Therefore, based on the protocol of Figure 3.2, the mediation of Prosper monitor is not guaranteed. From Assumption 2 we know that the monitor is not able to ensure *SAFE*. □

At this point the problem to be addressed is whether a hypercall is raised whenever a sensitive-operation for *SAFE* is going to be performed. To answer to this question, we analyse what are the security-sensitive operations. Two types of operations can modify the working set:

1. Operations that change the memory layout setting a physical block as executable.

2. Operations that modify the content of a physical block that is executable.

An operation that changes the memory layout raises a hypercall. This is due to the complete control of the hypervisor over the system memory. Consequently, all operations of the first type can be intercepted by the Prosper monitor and validated.

The second type of operations, namely operations that modify the content of an executable physical block, cannot be intercepted since they are not supposed to raise any hypercall. In fact, a simple write operation does not require the hypervisor intermediation since no modification of the memory layout is introduced. Without the interception of the second type of operation the Prosper monitor is not able to enforce the security property *SAFE* according to Proposition 1. In fact, a code injection attack can be performed without any interception by the monitor, only by writing malicious code in an executable area of the memory.

In order to address this problem a solution is to enforce an additional property over the memory layout to *avoid* all operations of the second type. In particular the property to be enforced is known in literature as *executable space protection* [23] [24]. This property states that, given a memory layout, a physical block cannot be both writable and executable. We formalize the concept of executable space protection over a system state σ with Definition 8 whereas with Definition 9 we introduce the security property $W \oplus X$similarly to how the property *SAFE* is introduced in Definition 6.

Definition 8 (**Property** $W \oplus X(\sigma)$)**.**
Given an ARM machine state $\sigma = \langle m, regs, psrs, coregs, mem \rangle$*:*

$$\begin{aligned} W \oplus X(\sigma) \equiv & \forall (va, pa, ex, w, r)(va', pa', ex', r', w'). \\ & (\langle pa, (ex, w, r) \rangle = mmu(\sigma, va) \wedge \langle pa', (ex', w', r') \rangle = mmu(\sigma, va')) \\ & \Rightarrow ((\exists pb.\ pa \in pb \wedge pa' \in pb) \Rightarrow \neg(w \wedge ex')) \end{aligned}$$

where pb is a physical block reference.

Definition 9 (**Property** $W \oplus X$)**.**
The (top level specification of the) system depicted in Figure 3.1 satisfies the property $W \oplus X$ *if and only if, given a machine state* σ *such that* $W \oplus X(\sigma)$ *holds, if* σ' *is a reachable state from* σ *then* $W \oplus X(\sigma')$ *holds.*

Assuming that the property $W \oplus X$ holds, since the operations 2) are not possible, the security-sensitive operations for *SAFE* are only those operations that change the memory layout by setting a physical block to executable. Since these operations always raise a hypercall, the necessary condition for the complete mediation stated by Proposition 1 is satisfied.

The Prosper monitor and the $W \oplus X$ property

The security property $W \oplus X$ belongs to the class of safety properties [5]. We also observe that $W \oplus X$ constrains the system memory layout since $W \oplus X(\sigma)$ states a condition over the access permissions to the physical blocks of σ. Therefore, the property $W \oplus X$ identified satisfies the necessary condition of *enforceable properties for Prosper monitor* stated by Fact 3. We can try to identify a suitable validation mechanism for the Prosper monitor to ensure this property.

A simple security policy to ensure $W \oplus X$ is the validation of all the security-sensitive operations for this property. The security sensitive operations for $W \oplus X$ can be identified as those that change the memory configuration. Since guests can only modify the memory configuration through the hypercalls, we are able to identify the security-sensitive operations for $W \oplus X$with the hypercalls, as stated by Definition 10.

Definition 10 (security-sensitive operations for $W \oplus X$).
An operation op is security-sensitive for the property $W \oplus X$ if and only if op is a hypercall.

The interaction protocol presented in Figure 3.2 guarantees that the Prosper monitor is informed about all the hypercalls. Consequently, the mediation of the monitor is guaranteed for all the security-sensitive operations for both $W \oplus X$ and *SAFE* and the security policy simply consists of the validation of the hypercalls, to ensure that the operation does not violate $W \oplus X$ or *SAFE*.

Validation mechanism

In this subsection we report all the reasoning done to identify the checks to be performed by the validation mechanism. The goal of these checks is the evaluation of the security-sensitive operations for $W \oplus X$ and *SAFE*. These operations are the hypercalls as concluded in the previous paragraph. We introduce in Table 3.1

a high level definition of the hypercalls provided by the hypervisor to the Linux guest to manage the memory layout [2]. Let us first describe more formally the concept of *mapping* which is used extensively in the definition of the hypercalls.

Definition 11 (**mapping**).
Let m be a mapping described by a page table descriptor (see Figure 2.4). Then, the physical address $m.pa$ is accessible through the virtual address $m.va$ with access permissions $m.ex$, $m.w$ and $m.r$ that give the executable, writable and readable rights.

From Table 3.1, we can identify four categories for the hypercalls. These categories are taken into exam in order to identify the checks that the validation mechanism must perform over the hypercalls. The categories are:

1. *create*: this category includes the hypercalls that create a page table, namely $createL1(\sigma, base)$ and $createL2(\sigma, base)$. We denote by $create\_(\sigma, base)$ a general hypercall of this category.

2. *map*: this category includes the three hypercalls that map a page table entry, these are $mapL1Sec(\sigma, base, m)$, $mapL1PT(\sigma, base, m)$ and $mapL2(\sigma, base, m.$ We denote by $map\_(\sigma, base, m)$ a general hypercall of this category.

3. *unmap*: this category includes those hypercalls that remove a mapping from a page table entry, namely $unmapL1(\sigma, base, m)$ and $unmapL2(\sigma, base, m)$. We denote by $unmap\_(\sigma, base, m)$ a general hypercall of this category.

4. *free*: this category includes the two hypercalls $freeL1(\sigma, base)$ and $freeL2(\sigma, base)$. We denote by $free\_(\sigma, base)$ a general hypercall of this category.

The idea for the validation mechanism is that, given a hypercall h, this operation is validated and accepted if and only if it is "secure" for both the properties $SAFE$ and $W \oplus X$. Definition 12 introduces the concept of secure hypercall.

Definition 12 (**secure hypercall**).
Let h be a hypercall, and let σ and σ' such that σ' is the reachable state from σ by

[2]We would like to observe that a hypercall modifies one page table at a time. This observation will be useful when the formal proof is discussed in Section 3.3

Hypecall	**Description**
1) $createL1(\sigma, base)$	Activates a first level page table (L1) starting from the physical address *base* in the state σ
2) $createL2(\sigma, base)$	Activates a second level page table (L2) starting from the physical address *base* in the state σ
3) $mapL1Sec(\sigma, base, m)$	Introduces a new mapping m in the L1 page table pointed by *base*. The mapping is between the virtual address $m.va$ and the physical address $m.pa$, with access permissions $(m.ex, m.w, m.r)$
4) $mapL1PT(\sigma, base, m)$	Introduces a new mapping m in the L1 page table pointed by *base*. The mapping is between the virtual address $m.va$ and the second level page table (L2) that starts from the physical address $m.pa$
5) $mapL2(\sigma, base, m)$	Introduces a new mapping m in the L2 page table pointed by *base*. The mapping is between the virtual address $m.va$ and the physical address $m.pa$ with access permissions $(m.w, m.r, m.ex)$
6) $unmapL1(\sigma, base, m)$	Removes the mapping of the virtual address $m.va$ from the L1 page table pointed by *base*
7) $unmapL2(\sigma, base, m)$	removes the mapping of the virtual address $m.va$ from the L2 page table pointed by *base*
8) $freeL1(\sigma, base)$	Deactivates the L1 page table pointed by *base*
9) $freeL2(\sigma, base)$	Deactivates the L2 page table pointed by *base*

Table 3.1: Hypercalls descritpion

performing the hypercall h. Assume that $safe(\sigma)$ and $W \oplus X(\sigma)$ hold. Then:

$$secure(h, \sigma) \Leftrightarrow secure_{W \oplus X}(h, \sigma) \wedge secure_{safe}(h, \sigma) \text{ where:}$$
$$secure_{W \oplus X}(h, \sigma) \Leftrightarrow W \oplus X(\sigma')$$
$$secure_{safe}(h, \sigma) \Leftrightarrow safe(\sigma')$$

Definition 12 states that a hypercall h is considered "secure" in a state if and only if, the state σ' that is reached after performing h satisfies the properties $safe(\sigma')$ and $W \oplus X(\sigma')$.

Given a hypercall h and a machine state σ, the validation mechanism needs to establish if $secure(h, \sigma)$ holds. If it holds, h is accepted, otherwise it is rejected. The approach adopted is the analysis of each category of hypercalls looking for *sufficient conditions* to conclude $secure(h, \sigma)$. We organize the discussion taking into analysis first the property $W \oplus X$ and then the property $SAFE$.

Validation mechanism for $W \oplus X$

In the following paragraphs we analyse how to establish the property $secure_{W \oplus X}(h, \sigma)$ given a hypercall h and a machine state σ. The goal is to identify a sufficient condition to conclude $secure_{W \oplus X}(h, \sigma)$. We examine each category of hypercall.

Category map: Let's take a hypercall h of type *map* and let σ be a machine state. To understand how to validate h, we try to establish when $W \oplus X(\sigma')$ holds. Based on Definition 8, $W \oplus X(\sigma')$ holds if and only if all the physical blocks of σ' are not both writable and executable. Clearly, the approach of checking all the page tables to establish if a given physical block respects the above condition after the mapping is not efficient. Moreover, each hypercall modifies only one page table at a time.

We can restrict the checks only to the page table that is subject of the hypercall: h introduces a mapping in the current master page table making a physical address $m.pa$ accessible with the permissions $(m.ex, m.w, m.r)$. Therefore, a minimal check to be performed over the access permissions is to ensure that the writable access and the executable access are not granted contemporary. This is expressed with the Fact 4.

Fact 4 (**necessary condition for the map hypercall**).
Let $h = map\_(\sigma, base, m)$. *Assume that* $W \oplus X(\sigma)$, *then:*

$$secure_{W \oplus X}(h, \sigma) \Rightarrow \neg(m.w \wedge m.ex)$$

However, the simple condition expressed by Fact 4 is only necessary and not sufficient for $secure_{W \oplus X}(h, \sigma)$. In fact, even if h specifies sound access permissions to the physical address $m.pa \in pb$, the $W \oplus X$ property can be violated in σ': suppose that the mapping m specified by h is such that $m.w = 1$, there might exist a different mapping m' (in some page table of σ) that grants an executable access, so $m'.ex = 1$, to a physical address $m'.pa$ such that $m'.pa \in pb$.

The solution identified to this problem exploits the infrastructure provided by the hypervisor. From Section 2.2.3, we know that the hypervisor maintains *reference counters* that track, for each physical block, the number of page table entries (L1 or L2) that point to that physical block and give the writable permission. This is done by the virtualization mechanism (see the paragraph titled *virtualization mechanism* of Section 2.2.3) of the hypervisor in order to write-protect the physical blocks containing page tables, and to constrain a guest to modify in user mode only physical blocks of type data. We decide to extend the reference counters of the hypervisor by counting also, for each physical block, the page table entries that point to that physical block and give the executable permission. Therefore, we distinguish two functions for the reference counting:

$$pgrefs_w : [0, 2^{20}) \longrightarrow \mathbb{N}$$
$$pgrefs_{ex} : [0, 2^{20}) \longrightarrow \mathbb{N}$$

Thanks to this extension, we are able to formalize a sufficient condition for $secure_{W \oplus X}(h, \sigma)$ with Fact 5, but first we need to introduce the concept of *sound mapping*:

Definition 13 (**sound mapping for** $W \oplus X$).
Let m *be a mapping and let* pb *be the physical block reference such that* $m.pa \in pb$. *Then:*

$$\begin{aligned} sound_{W \oplus X}(m) \Leftrightarrow & (m.w \Rightarrow \neg(m.ex) \wedge pgrefs_{ex}(pb) = 0) \wedge \\ & (m.ex \Rightarrow \neg(m.w) \wedge pgrefs_w(pb) = 0) \end{aligned}$$

A mapping m is sound if and only if, if the mapping m grants a writable access to the physical address $m.pa$, then the executable access is not granted and the physical block of $m.pa$ has the executable reference counter equal to zero. Similarly, if the mapping m grants an executable access to the physical address $m.pa$, then the writable access is not granted and the physical block of $m.pa$ has the writable reference counter equal to zero.

Fact 5 (**sufficient condition for the map hypercall**).
Let $h = map\_(\sigma, base, m)$. Assume that $W \oplus X(\sigma)$, then:

$$sound_{W \oplus X}(m) \Rightarrow secure_{W \oplus X}(h, \sigma)$$

Fact 5 states that, if the mapping introduced by the hypercall h is sound, then $secure_{W \oplus X}(h, \sigma)$ holds.

Category create: Let's take a hypercall h of type *create*. This changes the memory layout configuration by the activation of all the mappings of a page table. Conceptually, a page table creation can be seen as a sequence of hypercalls of type *map*. Therefore the validation mechanism can exploit Fact 5 to validate the hypercall h. However this is not sufficient since $pgrefs_{ex}$ and $pgrefs_{w}$ track the reference counters in the system state σ [3]. It is then necessary to make a cross-checking over all the mappings of the created page table. For this purpose we define the concept of mutually secure mappings with Definition 14.

Definition 14 (**mutually-secure mappings**).
Let m and m' be two mappings. Then:

$$\begin{aligned} mutually\_secure(m, m') \Leftrightarrow (\exists pb.\ m.pa \in pb \wedge m'.pa \in pb) \Rightarrow \\ (m.ex \Rightarrow \neg m'.w) \wedge (m'.ex \Rightarrow \neg m.w) \end{aligned}$$

where pb references a physical block.

Two mappings m and m' are mutually-secure if and only if, if they grant an access to the same physical block pb then, if m gives an executable right m' does

[3] The $pgrefs_{w}$ and $pgrefs_{ex}$ are part of the hypervisor state. They are managed only by the hypervisor and updated only after that a hypercall has been accepted by the monitor

not give the writable access, and if m' gives an executable access m does not give the writable access.

The sufficient condition for $secure_{W \oplus X}(h, \sigma)$ with h of type create is given with Fact 6

Fact 6 (**sufficient condition for the create hypercall**).
Let $h = create\_(\sigma, base)$ and let pt [4] be the page table pointed by base. Assume that $W \oplus X(\sigma)$, then:

$$(\forall m, m' \in pt.\ sound_{W \oplus X}(m) \wedge mutually\_secure(m, m')) \Rightarrow secure_{W \oplus X}(h, \sigma)$$

Fact 6 states that $secure_{W \oplus X}(h)$ holds if for each mapping m of the page table pt created by h, the mapping m is sound, and for each mapping m' of pt, m and m' are mutually-secure.

Category unmap: Let's take a hypercall h of type *unmap*. This hypercall changes the memory layout removing a mapping between a virtual and a physical address from the current master page table. Since this operation does not give any additional right of access to any physical block, it is considered a secure hypercall. Therefore, Fact 7 states that $secure_{W \oplus X}(h)$ holds without any checks.

Fact 7 (**sufficient condition for the unmap hypercall**).
Let $h = unmap\_(\sigma, base, m)$. Assume that $W \oplus X(\sigma)$, then:

$$secure_{W \oplus X}(h, \sigma)\ holds$$

Category free: Let's take a hypercall h of type *free*. This changes the memory layout configuration deactivating all the mappings of a page table. Conceptually, h can be seen as a sequence of hypercalls of type *unmap*. Therefore, a hypercall of type $free$ is considered secure as stated by Fact 8.

[4] A page table can be seen as a set of mappings, each one is given by a page table descriptor of an entry: $pt = \{m_1, ..., m_n\}$

Fact 8 (**sufficient condition for the free hypercall**).
Let $h = free\_(\sigma, base)$ *. Assume that* $W \oplus X(\sigma)$*, then:*

$$secure_{W \oplus X}(h, \sigma) \text{ holds}$$

Validation mechanism for SAFE

In the following paragraphs we analyse how to establish if $secure_{safe}(h, \sigma)$ holds (see Definition 12 for $secure_{safe}(h, \sigma)$) given a hypercall h and a machine state σ. The goal is to identify a sufficient condition C to conclude $secure_{safe}(h, \sigma)$. We analyse each category of hypercall.

Category map: Let h be a hypercall of type *map* and let σ be a machine state. This hypercall introduces a mapping for the physical address $m.pa$. If m grants an executable access to the physical block pb such that $m.pa \in pb$, then h would change the system working set $WS(\sigma)$ (see Definition 4 of $WS(\sigma)$). Therefore the hypercall h is secure if the executable code in pb has a valid signature according to the monitor golden image GI (see Definition 3 of golden image and the predicate *valid*). Based on this analysis we formalize a sufficient condition with Fact 9, but before that, we introduce the concept of *sound mapping* for the property *SAFE* with Definition 15.

Definition 15 (**sound mapping for SAFE**).
Let m *be a mapping and let* pb *be the physical block reference such that* $m.pa \in pb$*. Let* GI *be the golden image. Then:*

$$sound_{safe}(m, \sigma, GI) \Leftrightarrow (m.ex \Rightarrow valid(pb, \sigma, GI))$$

Definition 15 states that a mapping m is sound for *SAFE* if and only if, when granting an executable access to a physical block pb, then the physical block pb has a valid signature according to the golden image GI

Fact 9 (**sufficient condition for the map hypercall**).
Let $h = map\_(\sigma, base, m)$ *and let* GI *be the golden image. Assume that* $safe(\sigma)$*, then:*

$$sound_{safe}(m, \sigma, GI) \Rightarrow \ secure_{safe}(h, \sigma)$$

Fact 9 states that if the mapping m is sound according to Definition 15, then $secure_{safe}(h, \sigma)$ holds.

Category create: Let h be a hypercall of type *create*. This can be seen as a sequence of hypercalls of type *map* as explained in the previous paragraphs, when h is analysed to establish $secure_{W \oplus X}(h, \sigma)$. Consequently $secure_{safe}(h, \sigma)$ holds if all the mappings activated by the hypercall h are sound according to Definition 15. This is stated by Fact 10.

Fact 10 (**sufficient condition for the create hypercall**).
Let $h = create\_(\sigma, base)$ *and let* pt [5] *be the page table pointed by base. Let* GI *be the golden image. Assume that* $safe(\sigma)$*, then:*

$$(\forall m \in pt.\ sound_{safe}(m, \sigma, GI)) \Rightarrow secure_{safe}(h, \sigma)$$

Category unmap: Let's take a hypercall h of type *unmap*. This hypercall changes the memory layout removing a mapping between a virtual and a physical address from the current master page table. Since this operation does not introduce any new executable code in the working set, it is considered a secure hypercall. Therefore, Fact 11 states that $secure_{safe}(h, \sigma)$ holds without any checks.

Fact 11 (**sufficient condition for the unmap hypercall**).
Let $h = unmap\_(\sigma, base, m)$*. Assume that* $safe(\sigma)$*, then:*

$$secure_{safe}(h, \sigma)\ holds$$

Category free: Let's take a hypercall h of type *free*. This changes the memory layout configuration deactivating all the mappings of a page table. Conceptually, h can be seen as a sequence of hypercalls of type *unmap*. Therefore, a hypercall of type $free$ is considered secure as stated by Fact 12.

[5] A page table can be seen as a set of mappings, each one is given by a page table descriptor: $pt = \{m_1, ..., m_n\}$

Fact 12 (**sufficient condition for the free hypercall**).
Let $h = free\_(\sigma, base)$. *Assume that* $safe(\sigma)$, *then:*

$$secure_{safe}(h, \sigma) \text{ holds}$$

We are finally able to give a description of the Prosper monitor validation mechanism.

Definition 16 (**The Prosper runtime's validation mechanism**).
Let h *be a hypercall, and let* GI *be the golden image. Then:*

$$\begin{aligned}
validation(h) \equiv \\
&(h = map_(\sigma, base, m) \Rightarrow sound_{W \oplus X}(m) \wedge sound_{safe}(m, \sigma, GI)) \wedge \\
&(h = create_(\sigma, base) \Rightarrow \forall m, m' \in pt^{6}.\ sound_{W \oplus X}(m) \wedge sound_{safe}(m, \sigma, GI) \wedge \\
&\qquad\qquad mutually_secure(m, m')) \wedge \\
&(h = unmap_(\sigma, base) \Rightarrow true) \wedge \\
&(h = free_(\sigma, base) \Rightarrow \quad true)
\end{aligned}$$

The validation mechanism of a hypercall h is a predicate $validate(h)$ that holds if and only if h satisfies the sufficient conditions discussed in the previous paragraphs (see Fact 5 - 11).

3.3 Formal proof of correctness

In this section we present the formal proof of correctness for the validation mechanism discussed in Section 3.2.3. We would like to demonstrate that the validation mechanism defined with Definition 16 guarantees the security properties *SAFE* and $W \oplus X$.

First we present a top level specification of the behaviour of the system shown in Figure 3.1, This is based on the specification described in Section 2.2.3, where

[6] pt is the page table pointed by base

the Prosper hypervisor is presented. Then, we formalize the goals of our proof as theorems. For each theorem, we will discuss a possible proof presenting all the intermediate lemmas. The proofs discussed has been verified in the HOL4 theorem prover. In Chapter 4 we will discuss the proof verification with the HOL4.

3.3.1 Top Level Specification

In order to give a top level specification of the system shown in Figure 3.1, we distinguish three components:

- An ARM machine with a state $\sigma \in \Sigma$ such that $\sigma = \langle m, regs, psrs, coregs, mem \rangle$ (see Section 2.2.2 where the formal model of an ARM machine is defined).
- The hypervisor, with an abstract state η that represents the data structures for the page typing and the reference counting: $pgrefs_{ex}$, $pgrefs_{w}$ and $pgtype$.
- The runtime monitor, with an abstract state GI that consists of the golden image.

Similarly to the formalization introduced in Section 2.2.3 (see the paragraph titled *Top Level Specification*), we define the behaviour of our system with a transition system that is described by a top level specification (TLS).

The system behaviour alternates executions in user mode, in which neither the hypervisor nor the monitor intermediation is required, with executions in privileged mode. The executions in privileged mode require the hypervisor intermediation as stated by the inference rule *priv* of the TLS in Section 2.2.3. The intermediation is invoked by a hypercall that triggers a hypervisor handler H_a. In these executions, based on the interaction protocol of Figure 3.2, the monitor invocation is also guaranteed. The monitor validates the hypercall based on what stated with Definition 16.

$$\frac{\langle \sigma, \eta \rangle \xrightarrow{op}_0 \langle \sigma', \eta' \rangle}{\langle \sigma, \eta, GI \rangle \overset{\text{op}}{\hookrightarrow}_0 \langle \sigma', \eta', GI \rangle}\ usr' \qquad \frac{\langle \sigma, \eta \rangle \xrightarrow{op}_1 \langle \sigma', \eta' \rangle \qquad validate(op)}{\langle \sigma, \eta, GI \rangle \overset{\text{op}}{\hookrightarrow}_1 \langle \sigma', \eta', GI \rangle}\ securepriv$$

$$\frac{\langle \sigma, \eta \rangle \xrightarrow{op}_1 \langle \sigma', \eta' \rangle \qquad \neg validate(op)}{\langle \sigma, \eta, GI \rangle \overset{\text{err}}{\hookrightarrow}_1 \xi(\langle \sigma, \eta, GI \rangle)}\ insecurepriv$$

The inference rule usr' states that, executions in user mode behave exactly as specified by the inference rule usr of the TLS in section 2.2.3, without affecting the monitor. The inference rule $securepriv$ describes the system behaviour in case of a secure privileged execution: if op is validated by the monitor and $validate(h)$ holds, the system behaves as described by the inference rule $priv$. In case the privileged operation op is such that $validate(h)$ does not hold, the system performs a special operation err reaching a state that is returned by a function ξ. The function ξ handles the case in which the operation requested op is not consistent with the property $SAFE$ or $W \oplus X$. In this thesis, and for all the following proofs we consider that ξ is the identity function and the operation err is the empty operation denoted by τ. Therefore, the inference rule $insecurepriv$ becomes:

$$\frac{\langle \sigma, \eta \rangle \xrightarrow{op}_1 \langle \sigma', \eta' \rangle \qquad \neg validate(op)}{\langle \sigma, \eta, GI \rangle \overset{\tau}{\hookrightarrow}_1 \langle \sigma, \eta, GI \rangle} \; insecurepriv$$

The top level specification presented fully describes the behaviour of our system: the only transitions that our system performs are those described by the rules usr, $securepriv$ and $insecurepriv$.

3.3.2 Goals formalization and proof

The main goal of the Propsper monitor is to ensure the security property $SAFE$ in order to protect the Linux kernel guest from code injection attacks. We know from the discussion of Section 3.2.3 that the property $W \oplus X$must be enforced as well. To prove that the validation mechanism ensures the properties, we formalize the goals with the Theorems 4 and 5 for the property $SAFE$, and Theorem 6 for the property $W \oplus X$. Before that, first we introduce the following sets:

- Let Q_{safe} be the set of system states $\langle \sigma, \eta, GI \rangle$ such that $safe(\sigma)$ holds (See definition 5) .
- Let $Q_{W \oplus X}$ be the set of system states $\langle \sigma, \eta, GI \rangle$ such that $W \oplus X(\sigma)$ holds (See Definition 8) .
- Let Q_{Inv} be the set of system states $\langle \sigma, \eta, GI \rangle$ such that $Inv(\langle \sigma, h, m \rangle)$ holds, where Inv is a system state invariant discussed and presented in the next paragraphs (see Definition 18).

- Let $Q = Q_{safe} \cap Q_{W \oplus X} \cap Q_{Inv}$ be the intersection set. We refer to a system state in Q as a *consistent* system state.

Theorem 4 (*SAFE* over privileged transitions).
Let $\langle \sigma, \eta, GI \rangle \in Q$. *Then, if* $\langle \sigma, \eta, GI \rangle \overset{op}{\hookrightarrow}_1 \langle \sigma', \eta', GI \rangle$ *then* $\langle \sigma', \eta', GI \rangle \in Q_{safe}$

Theorem 4 states that starting from a consistent system state, the property *safe* is satisfied by the reached state after a privileged transition.

Proof. We must show that $safe(\sigma')$ holds. From the assumption we have that $\langle \sigma, \eta, GI \rangle \in Q$, therefore $\langle \sigma, \eta, GI \rangle \in Q_{safe}$. We can distinguish two cases:

1. $op = \tau$: then, based on the inference rule *insecurepriv*, we have that $\langle \sigma, \eta, GI \rangle = \langle \sigma', \eta', GI \rangle$. Since $\langle \sigma, \eta, GI \rangle \in Q_{safe}$ it follows that $\langle \sigma', \eta', GI \rangle \in Q_{safe}$ and $safe(\sigma')$ holds.

2. $op \neq \tau$: since the transition considered is privileged, *op* must be a hypercall. Let h be such that $op = h$. Based on the inference rule *securepriv*, we know that $validate(h)$ holds and h satisfies the sufficient conditions identified in Section 3.2.3 (see Definition 16 of $validate(h)$). To prove $safe(\sigma')$, we can proceed by cases based on the category of the hypercall h and using the sufficient conditions identified:

   - $h = map\_(\sigma, base, m)$: then, since $validate(h)$ holds, we have that $sound_{safe}(m, \sigma, GI)$ (see Definition 15 for $sound_{safe}$). We distinguish two cases based on whether m grants an executable access to a physical block or not:

     – $m.ex = 0$: in this case the hypercall h does not change the working set, namely $WS(\sigma) = WS(\sigma')$. Therefore, since $safe(\sigma)$ holds, it follows that $safe(\sigma')$ holds as well (notice that $safe(\sigma)$ states a condition over the working set, if this set does not change, the condition is still valid. See Definition 5 for $safe(\sigma)$).

     – $m.ex = 1$: in this case, m is granting an executable access to the physical block pb such that $m.pa \in pb$. Therefore, we have a new element in the working set: $WS(\sigma') = WS(\sigma) \cup \{\langle pb, content(pb, \sigma) \rangle\}$. To prove that $safe(\sigma')$ holds, it is sufficient to prove that the new

element of the working set $\langle pb, content(pb, \sigma)\rangle$ has a valid signature according to the golden image GI. Since $sound_{safe}(m, \sigma, GI)$, it follows that $valid(pb, \sigma, GI)$ holds. Therefore $safe(\sigma')$ holds.

- $h = create\_(\sigma, base)$: then, since $validate(h)$ holds, we have that: $\forall m \in pt$[7]$.sound_{safe}(m, \sigma, GI)$. For each mapping m of the created page table pt, we know that m is sound. We can proceed as in the previous case showing that, if m grants an executable access to a physical block pb, pb has a valid signature according to the golden image GI. Therfore, we can conclude that $safe(\sigma')$ holds.
- $h = unmap\_(\sigma, m, base)$: we distinguish two cases to prove that $safe(\sigma')$ holds:
  - $m.ex = 0$, then $WS(\sigma) = WS(\sigma')$. Since $safe(\sigma)$ holds, it follows that $safe(\sigma')$ holds.
  - $m.ex = 1$, we have that $WS(\sigma') = WS(\sigma) \setminus \{\langle pb, content(pb, \sigma)\rangle\}$, where pb is such that $m.pa \in pb$. Since $WS(\sigma') \subset WS(\sigma)$, it follows that $safe(\sigma')$ holds. In fact, each element of $WS(\sigma')$ is an element of $WS(\sigma)$ with a valid signature according to GI.
- $h = free\_(\sigma, base)$: similarly to the previous case, we can state that $WS(\sigma') \subseteq WS(\sigma)$. Consequently, from $safe(\sigma)$ we can conclude that $safe(\sigma')$ holds.

□

Important observation: It is worth to notice that, using the sufficient conditions identified and formalized with Facts 9 - 12 we are able to prove the Theorem 4. This observation allows us to conclude that our sufficient conditions are effective and "strong" enough to ensure the property $SAFE$ over the privileged transitions.

Theorem 5 (*SAFE* over user transitions).
Let $\langle \sigma, \eta, GI\rangle \in Q$. *Then, if* $\langle \sigma, \eta, GI\rangle \overset{op}{\hookrightarrow}_0 \langle \sigma', \eta', GI\rangle$ *then* $\langle \sigma', \eta', GI\rangle \in Q_{safe}$

[7] pt is the page table pointed by base

Theorem 5 states that starting from a consistent system state, the property *safe* is satisfied by the reached state after a user transition. Notice that a transition in user mode is performed without the monitor intermediation. Therefore, giving a proof for Theorem 5, implies that the validation mechanism of the monitor is strong, and effective enough, to enforce a correct system state over all the possible transitions of the system. The proof of Theorem 5 depends on the Lemma 1 and the Corollary 1. Therefore, we suggest to read first the Corollary 1 and then the Lemma 1 before reading the following proof.

Proof. Based on the hypothesis $\langle \sigma, \eta, GI \rangle \in Q$, it follows that $\langle \sigma, \eta, GI \rangle \in Q_{safe}$, and $safe(\sigma)$ holds. The Lemma 1 states that $WS(\sigma) = WS(\sigma')$. Therefore, by Definition 5 of $safe(\sigma)$, we conclude that $safe(\sigma')$ holds, and $\langle \sigma, \eta, GI \rangle \in Q_{safe}$. □

The following corollary is a consequence of Theorem 1 that is related to the hypervisor isolation properties (see Section 2.2, the paragraph titled *Isolation Property*). Theorem 1 states that the hypervisor invariant I is satisfied by all the states reached by the system running on top of the hypervisor.

Corollary 1 (**Isolation properties consequence**).
Let $\langle \sigma, \eta, GI \rangle \in Q$ *. Assume that* $\langle \sigma, \eta, GI \rangle \overset{op}{\hookrightarrow}_0 \langle \sigma', \eta', GI \rangle$*. Then, given a physical block referenced by pb, if* $content(pb, \sigma) \neq content(pb, \sigma')$ *then* $\eta.pgtype(pb) = D$*.*

The Corollary 1 states that, only the physical blocks of type data can change their content when a user transition is performed by the system.

Proof. by Theorem 1 we know that the hypervisor invariant I is preserved over all the system states $\langle \sigma, \eta \rangle$ of the TLS described in Section 2.2.3. The invariant I states, among other things, that a page table descriptor grants a writable access to a physical block *pb* if and only if *pb* is of type data ($\eta.pgtype(pb) = D$). Consequently if a physical block has a different content after a user transition, this block can only be of type data. Considering that the monitor is a guest of the hypervisor (see Figure 3.1), we can conclude that the invariant I is preserved also over all the system states $\langle \sigma, \eta, GI \rangle$. Therefore, only physical blocks of type data can change their content after a user transition of our system. □

Lemma 1 (**Equal working sets over user transitions**).
Let $\langle \sigma, \eta, GI \rangle \in Q$*. If* $\langle \sigma, \eta, GI \rangle \overset{op}{\hookrightarrow}_0 \langle \sigma', \eta', GI \rangle$ *then* $WS(\sigma) = WS(\sigma')$

Lemma 1 states that system working set does not change after a user transition.

Proof. To prove that $WS(\sigma) = WS(\sigma')$, first we prove that $WS(\sigma) \subseteq WS(\sigma')$, then we prove that $WS(\sigma') \subseteq WS(\sigma)$.

- Let $\langle pb, c\rangle \in WS(\sigma)$ and suppose that $\langle pb, c\rangle \notin WS(\sigma')$. Since $\langle pb, c\rangle \in WS(\sigma)$, there exists a mapping m in the state $\langle \sigma, \eta, GI\rangle$ such that $m.ex = 1$ and $m.pa \in pb$. If we assume that $\langle pb, c\rangle \notin WS(\sigma')$ we can identify two possible cases (and they are the only possible cases):

  - The mapping m has been removed from the system with the operation op, which means that a page table descriptor has been modified. Let pb' be the reference to the physical block containing this page table. Removing m from the page table implies that $content(pb', \sigma) \neq content(pb', \sigma')$ and $\eta.pgtype(pb') = L1 \vee \eta.pgtype(pb') = L2$, namely $\eta.pgtype(pb') \neq D$. This is in contradiction with Corollary 1. Therefore, we can conclude that $\langle pb, content(pb, \sigma)\rangle \in WS(\sigma')$
  - The mapping m is still active in the system state $\langle \sigma', \eta', GI\rangle$. If we assume that $\langle pb, c\rangle \notin WS(\sigma')$, it means that $content(pb, \sigma) \neq content(pb, \sigma')$ (since the physical block pb is still executable thanks to m, the only possibility is that the content of pb is different in σ'). If the operation pb is able to change the content of pb, it means that there exists a mapping m' in the system state $\langle \sigma, \eta, GI\rangle$ such that $m'.w = 1$ and $m'.pa \in pb$. But this means that pb is a physical block both writable and executable. This is in contradiction with our assumptions. In fact if $\langle \sigma, \eta, GI\rangle \in Q$ then $\langle \sigma, \eta, GI\rangle \in Q_{W\oplus X}$ and a physical block cannot be both writable and executable. Therefore, we conclude that $\langle pb, c\rangle \in WS(\sigma')$.

- The proof $WS(\sigma') \subseteq WS(\sigma)$ is similar to the previous case.

□

Theorem 6 ($\mathbf{W \oplus X}$ over the system transitions).
Let $\langle \sigma, \eta, GI\rangle \in Q$ and $i \in \{0, 1\}$. Then, if $\langle \sigma, \eta, GI\rangle \overset{op}{\hookrightarrow}_i \langle \sigma', \eta', GI\rangle$, then $\langle \sigma', \eta', GI\rangle \in Q_{W\oplus X}$.

The Theorem 6 states that starting from a consistent system state, the property $W \oplus X$ is satisfied by the reached state after a system transition (user and privileged).

The proof of Theorem 6 is the most challenging in all this formal verification. We adopt the following strategy for our proof:

1. We introduce a property *Inv*, that is defined with Definition 17
2. We show that *Inv* is an *invariant* property. This means that *Inv* holds over all the system states. This is formalized and proved with the Lemmas 2 and 3.
3. We show that, if a system state satisfies *Inv* then it satisfies $W \oplus X$. This is formalized and proved with Theorem 7.

Definition 17 (**The monitor invariant *Inv*).**
Let $\langle \sigma, \eta, GI \rangle$ *be a system state. Then:*

$$Inv(\langle \sigma, \eta, GI \rangle) \equiv \forall pb.\ (\eta.pgtype(pb) = L1\ \vee\ \eta.pgtype(pb) = L2) \Rightarrow \forall m \in pb.\ sound_{W \oplus X}(m)$$

In order to simplify the reading of the next proofs, we give another formulation of Definition 17 extending the definition of $sound_{W \oplus X}$ (that is defined with Definition 13).

Definition 18 (**The monitor invariant *Inv*).**
Let $\langle \sigma, \eta, GI \rangle$ *be a system state. Then*[6] *:*

$$\begin{aligned} Inv(\langle \sigma, \eta, GI \rangle) \equiv \forall pb.(\eta.\ \ & pgtype(pb) = L1\ \vee\ \eta.pgtype(pb) = L2) \Rightarrow \\ & \forall m \in pb.((m.ex \Rightarrow \neg m.w \wedge \eta.pgrefs_w(m.pb) = 0) \wedge \\ & (m.w \Rightarrow \neg m.ex \wedge\ \eta.pgrefs_{ex}(m.pb) = 0)) \end{aligned}$$

[6]In this definition, we use the notation $m.pb$ to refer the physical block pb such that $m.pa \in pb$

Definition 18 states that, the invariant Inv holds in a system state if and only if, for each physical block pb, if pb is of type $L1$ or $L2$ (meaning that pb contains a page table pt) then, for each mapping m of pt, if m grants an access to a physical block $m.pb$, this access cannot be both writable and executable. Furthermore, if m grants a writable access then the executable reference counter of $m.pb$ must be zero. Similarly, if m grants an executable access, then the writable reference counter of $m.pb$ must be zero.

Lemma 2 (*Inv* over user transitions).
Let $\langle \sigma, \eta, GI \rangle \in Q$. *Then, if* $\langle \sigma, \eta, GI \rangle \overset{op}{\hookrightarrow}_0 \langle \sigma', \eta', GI \rangle$ *then* $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$

Lemma 3 states that starting from a consistent system state, the invariant Inv is satisfied by the reached state after a user transition.

Proof. From the hypothesis we know that $\langle \sigma, \eta, GI \rangle \in Q_{Inv}$. Suppose that $\langle \sigma', \eta', GI \rangle \notin Q_{Inv}$. Then, there exists a physical block pb containing a page table pt with a mapping m (in the state $\langle \sigma', \eta', GI \rangle$) such that, $sound_{W \oplus X}(m)$ does not hold. If m is not sound, we can identify two possible cases (that arises from the negation of $sound_{W \oplus X}(m)$):

1. $m.ex = 1$: m grants an executable access to a physical block $m.pb$ that is both writable and executable (in fact, if m is not sound, either $m.w = 1$, or $pgrefs_w(m.pb) \neq 0$). However, since $m.pb$ cannot be both writable and executable in the system state $\langle \sigma, \eta, GI \rangle \in Q_{W \oplus X}$, this means that, with the operation op a new mapping is introduced. Therefore, there exists a physical block pb' such that $content(pb', \sigma) \neq content(pb', \sigma')$ and $(\eta.pgtype(pb') = L1 \vee \eta.pgtype(pb') = L2)$, namely $\eta.pgtype(pb') \neq D$. This is in contradiction with Corollary 1.

2. $m.w = 1$: m grants a writable access to a physical block $m.pb$ that is both writable and executable (in fact, if m is not sound, either $m.ex = 1$, or $pgrefs_{ex}(m.pb) \neq 0$). With a similar reasoning to the previous case, we can conclude that a new mapping is introduced with the operation op which is in contradiction with Corollary 1.

Since both cases are not possible, we conclude the proof stating that, a page table such as pt with a mapping such as m is not possible in the state $\langle \sigma', \eta', GI \rangle$, therefore $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$. □

Lemma 3 (*Inv* over privileged transitions).
Let $\langle \sigma, \eta, GI \rangle \in Q$. *Then, if* $\langle \sigma, \eta, GI \rangle \overset{op}{\hookrightarrow}_1 \langle \sigma', \eta', GI \rangle$ *then* $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$

Lemma 3 states that starting from a consistent system state, the invariant *Inv* is satisfied by the reached state after a privileged transition.

Proof. We must show that $Inv(\langle \sigma', \eta', GI \rangle)$ holds. From the assumption we have that $\langle \sigma, \eta, GI \rangle \in Q$, therefore $\langle \sigma, \eta, GI \rangle \in Q_{Inv}$. We can distinguish two cases:

1. $op = \tau$: then, based on the inference rule *insecurepriv*, we have that $\langle \sigma, \eta, GI \rangle = \langle \sigma', \eta', GI \rangle$. Since $\langle \sigma, \eta, GI \rangle \in Q_{Inv}$ it follows that $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$ and $Inv(\langle \sigma', \eta', GI \rangle)$ holds.

2. $op \neq \tau$: since the transition considered is privileged, *op* must be a hypercall. Let h be such that $op = h$. Based on the inference rule *securepriv*, we know that $validate(h)$ holds and h satisfies the sufficient conditions identified in Section 3.2.3 (see Definition 16 of $validate(h)$). To prove $Inv(\langle \sigma', \eta', GI \rangle)$, we can proceed by cases based on the category of the hypercall h and using the sufficient conditions identified for the validation mechanism:

   - $h = map\_(\sigma, base, m)$: we know that $validate(h)$ holds, this means that $sound_{W \oplus X}(m)$ holds. Suppose that $\langle \sigma', \eta', GI \rangle \notin Q_{Inv}$. Then, there exists a physical block *pb* containing a page table *pt* with a mapping m' (in the state $\langle \sigma', \eta', GI \rangle$) such that, $sound_{W \oplus X}(m')$ does not hold. We distinguish two possible cases:
     - $m = m'$: since m is sound and m' is not sound, we have a contradiction. Therefore, this case is not possible.
     - $m \neq m'$: since the operation h introduces only one mapping at a time, we conclude that m' is an active mapping in *pt* also in the state $\langle \sigma, \eta, GI \rangle$. Therefore, *pb* is a physical block that violates the invariant definition. This means that $\langle \sigma, \eta, GI \rangle \notin Q_{Inv}$, which is in contradiction with our hypothesis.

   We conclude that a physical block such as *pb* containing a mapping such as m' is not possible in the state $\langle \sigma', \eta', GI \rangle$. Therefore, $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$ and $Inv(\langle \sigma', \eta', GI \rangle)$ holds.

- $h = create\_(\sigma, base)$: let pt be the page table pointed by $base$ and pb the physical block containing pt. We know that $validate(h)$ holds. This means that all the mappings of pt are sound. Suppose that $\langle \sigma', \eta', GI \rangle \notin Q_{Inv}$. Then, there exists a physical block pb' containing a page table pt' with a mapping m' (in the state $\langle \sigma', \eta', GI \rangle$) such that, $sound_{W \oplus X}(m')$ does not hold. We distinguish two possible cases:
  - $pt = pt'$: since all the mappings of pt are sound, a mapping such as m' is not possible.
  - $pt \neq pt'$: since the operation h creates only one page table at a time, we conclude that pt' is an active page table also in the state $\langle \sigma, \eta, GI \rangle$. Therefore, pb' is a physical block that violates the invariant definition. This means that $\langle \sigma, \eta, GI \rangle \notin Q_{Inv}$, which is in contradiction with our hypothesis.

  We conclude that a physical block such as pb' containing a mapping such as m' is not possible in the state $\langle \sigma', \eta', GI \rangle$. Therefore, $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$ and $Inv(\langle \sigma', \eta', GI \rangle)$ holds.
- $h = unmap\_(\sigma, m, base)$: let pt be the page table pointed by $base$ and pb the physical block containing pt. Then, suppose that $\langle \sigma', \eta', GI \rangle \notin Q_{Inv}$. This means that there exists a physical block pb' containing a page table pt' with a mapping m' (in the state $\langle \sigma', \eta', GI \rangle$) such that, $sound_{W \oplus X}(m')$ does not hold. We distinguish two possible cases:
  - $pt = pt'$: since the operation h is not introducing any new mappings in pt, this means that m' is an active mapping also in the state $\langle \sigma, \eta, GI \rangle$, and that pb violates the invariant definition since m' is not sound. This means that $\langle \sigma, \eta, GI \rangle \notin Q_{Inv}$, which is in contradiction with our hypothesis.
  - $pt \neq pt'$: since the operation h changes only one page table at a time, we conclude that m' is is an active mapping in pt' in the state $\langle \sigma, \eta, GI \rangle$. Therefore, pb' is a physical block that violates the invariant definition since m' is not sound. This means that $\langle \sigma, \eta, GI \rangle \notin Q_{Inv}$, which is in contradiction with our hypothesis.

  We conclude that a physical block such as pb' containing a mapping such as m' is not possible in the state $\langle \sigma', \eta', GI \rangle$. Therefore, $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$ and $Inv(\langle \sigma', \eta', GI \rangle)$ holds.

- $h = \mathit{free\_}(\sigma, \mathit{base})$: with exactly the same reasoning followed for the hypercalls of type *unmap*, it is possible to conclude that $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$ and $Inv(\langle \sigma', \eta', GI \rangle)$ holds.

□

Theorem 7 (*Inv* implies W ⊕ X).
If $\langle \sigma, \eta, GI \rangle \in Q_{Inv}$ *then* $\langle \sigma, \eta, GI \rangle \in Q_{W \oplus X}$.

Theorem 7 states that, if a system state $\langle \sigma, \eta, GI \rangle$ satisfies the monitor invariant ($Inv(\langle \sigma, \eta, GI \rangle)$ holds), then it satisfies also the $W \oplus X$ property ($W \oplus X(\sigma)$ holds).

Proof. Suppose that $\langle \sigma, \eta, GI \rangle \in Q_{Inv}$ and $\langle \sigma, \eta, GI \rangle \notin Q_{W \oplus X}$, namely $W \oplus X(\sigma)$ does not hold. This means that (see Definition 8 for $W \oplus X(\sigma)$ definition):

$$\begin{aligned}\neg W \oplus X(\sigma) \equiv & \exists (va, pa, r, w, ex)\ (va', pa', r', w', ex'). \\ & (\langle pa, (r, w, ex) \rangle = mmu(\sigma, va) \wedge \langle pa', (r', w', ex') \rangle = mmu(\sigma, va')) \wedge \\ & (\exists pb.pa \in pb \wedge pa' \in pb) \wedge (w \wedge ex')\end{aligned}$$

In other words (using the concept of mapping), there exist two mappings m and m' and a physical block referenced by pb such that:

- $m.pa \in pb$ and $m.w = 1$
- $m'.pa \in pb$ and $m.ex = 1$

Since $m'.ex = 1$, this implies that $pgrefs_{ex}(pb) \neq 0$ (in particular, $pgrefs_{ex}(pb) > 0$). Therefore, $sound_{W \oplus X}(m)$ does not hold. Let pt be the page table such that $m \in pt$, and let pb' be the physical block (of type L1 or L2) containing the page table pt. Then pb' violates the invariant definition. This means that $\langle \sigma, \eta, GI \rangle \notin Q_{Inv}$ which is in contradiction with our hypothesis. We conclude then that $W \oplus X(\sigma)$ holds and $\langle \sigma, h, g \rangle \in Q_{W \oplus X}$.

□

Thanks to Theorem 7 the proof of Theorem 6 is trivial. In fact, starting from a consistent system state $\langle \sigma, \eta, GI \rangle \in Q$, if $\langle \sigma, \eta, GI \rangle \overset{op}{\hookrightarrow}_i \langle \sigma', \eta', GI \rangle$, then $\langle \sigma', \eta', GI \rangle \in Q_{Inv}$ (since *Inv* is an invariant). Therefore, based on Theorem 7, we know that $\langle \sigma', \eta', GI \rangle \in Q_{W \oplus X}$.

Important observation: It is worth to notice that, using the sufficient conditions identified and formalized with Facts 5 - 8 we have been able to prove the Theorem 6 adopting the strategy discussed in the previous paragraphs. This fact allows us to conclude that our sufficient conditions are effective and strong enough to ensure the property $W \oplus X$over all the system transitions. Therefore, the monitor's validation mechanism is able to ensure the security property $W \oplus X$.

Theorem 7 concludes the formal proof of correctness performed in this work. All the theorems and lemmas have been verified in the HOL4 theorem prover. The reader, if interested, can find an example of proof with the HOL4 in Section 4.2.

Chapter 4

Correctness of Prosper monitor with HOL4

The formal proof of Section 3.3 has been verified with the help of the proof assistant HOL4. Actually, the overall structure of the proof was developed in parallel with its verification with the HOL4.

4.1 General structure of the proof verification

The Prosper runtime monitor is formalized as a model named `monitor_model` in the HOL4 theorem prover (see Section 2.3 for an overview about Automated Theorem Proving and the HOL4). The model implements the validation mechanism described in Section 3.2.3 with Definition 16. The security properties *SAFE* and $W \oplus X$are formally verified on this model following the approach presented in Section 3.3 where the formal proof is sketched.

Many theories and libraries are used for the formal verification, here we mention only some of them: the `hypervisor_modelTheory` is a theory that includes the Prosper hypervisor model and the theorems related to the isolation properties. The theories `wordLib` and `blastTheory` [1] instead have been extensively used to manage values and proofs over the data type `word`.

The proofs have been performed adopting the *goal oriented methodology* (see Section 2.3) and using the interactive session of the HOL4 assistant. An interactive

session that assists in proving a theorem t holds a goal stack that arises from the goal oriented methodology. The theorem t is proved if and only if all the goals of the stack have been reduced. In our proofs the HOL4 structure `Tactic` has been extensively used. This structure provides several tactics that implement automatic reasoners. The `GEN_TAC`, for example, is a tactic that strips the outermost universal quantifier from the conclusion of a goal. When the `GEN_TAC` is applied, this tactic operates on the goal that is on top of the stack and the universally quantified variable is arbitrarily instantiated[1].

In the formal development we tried to factorize the code as much as possible. To this purpose, all the proofs have been studied before any encoding, and a strategy of demonstration has been identified for each theorem. Thanks to this approach, we have been able to identify a set of helper theorems and lemmas that can be used in several proofs. This is similar to what is normally done in a programming task, when the identification of the most used (and common) functionalities is necessary before the encoding.

Table 4.1, summarizes the size of the formal development that is about 5k LOC [2] in HOL4.

Prosper monitor model	710 LOC
Helper theorems	680 LOC
Invariant proof	3145 LOC
$W \oplus X$ user transitions	433 LOC
$W \oplus X$ privileged transitions	100 LOC

Table 4.1: LOC of HOL4 development

It is beyond the goal this thesis to present all the HOL4 code development of the verification of the Prosper monitor validation mechanism. However, to give the reader an idea of this activity, that is a significant part of the work on which this thesis is based, in the next section we describe the HOL4 code for the verification of a single lemma.

[1]The only purpose here is to give a general idea about the implementation work related to the formal proof of correctness that is a consistent part of this thesis

[2]LOC denotes line of code

4.2 Example of proof with the HOL4

In this section we present a proof's example in the HOL4 proof assistant for a lemma. Fist we present the definition of the lemma, then we show the implemented proof. The presented lemma has been used to proof the Theorem 7 discussed in Section 3.3.

Algorithm 1: Definition of the lemma

```
val lemma_w_xor_ex_property_def = Define ‘
lemma_w_xor_ex_property = !(c1:sctlrT) (c2:word32) (c3:word32)
(mem:(word32 -> word8)) (pgtype:word20->word3)
(pgrefs_w:word20->word30) (pgrefs_ex:word20->word30).
monitor_invariant c1 c2 c3 mem pgtype pgrefs_w pgrefs_ex ==>
 !phy_page:word20.
((~(pgrefs_w phy_page = 0w)) ==> (pgrefs_ex phy_page = 0w)) /\
((~(pgrefs_ex phy_page = 0w)) ==> (pgrefs_w phy_page = 0w))
‘;
```

The code[3] of Algorithm 1 introduces a definition of the lemma in the HOL4 theorem prover. The evaluation of the definition adds the lemma named `lemma_w_xor_ex_property` to the available concepts of the interactive session. The definition states that: for each value of the system state (i.e. the registers `c1` `c2` and `c3` of the coprocessor 15 that controls the MMU, the system memory `mem`, the page typing function `pgtype` and the functions for the reference counting `pgrefs_w` and `pgrefs_ex`), if the `monitor_invariant` holds in the system state then, for each physical block if it is writable (has the writable reference counter different from zero) it cannot be executable (the executable reference counter is zero), and vice-versa.

[3]In the HOL4 code the operator ! represents the operator $\forall$

Algorithm 2: Proof of the lemma

```
0.val lemma_w_xor_ex_property_thm  = prove( ``lemma_w_xor_ex_property``,
1.   FULL_SIMP_TAC(srw_ss())[lemma_w_xor_ex_property_def]
2.   THEN (REPEAT(GEN_TAC))
3.   THEN (STRIP_TAC)
4.   THEN (GEN_TAC)
5.   THEN (ASSUME_TAC (SPECL[``c1:sctlrT``,``c2:word32``,``c3:word32``,
6.        ``mem:word32->word8``,``pgtype:word20->word2``,
7.        ``pgrefs_w:word20->word30``, ``pgrefs_ex:word20->word30``]
8.          pgrefs_thm))
9.   THEN (FULL_SIMP_TAC(srw_ss())[invariant_ref_cnt_def])
10.  THEN (SPEC_ASSUMPTION_TAC ``!pg:word20.p`` ``phy_page:word20``)
11.  THEN (RW_TAC(srw_ss())[])
12.  THENL[
13.      `(((count_pages (mem:word32->word8) (pgtype:word20->word2)
14.      phy_page (λ (ap:word3) (xn:bool). ¬xn):num) = 0))`
15.      by (count_refs_is_zero)
16.      THEN (FULL_SIMP_TAC(srw_ss())[]),
17.      `((count_pages (mem:word32->word8) (pgtype:word20->word2)
18.      phy_page (λ (ap:word3) (xn:bool). ap = (3w:word3)):num) = 0)`
19.      by (count_refs_is_zero)
20.      THEN (FULL_SIMP_TAC(srw_ss())[])
21.    ]
22. );
```

The code in Algorithm 2 presents a goal oriented proof to the lemma defined in Algorithm 1. The interactive evaluation of the code returns a theorem that is added to the available concepts of the interactive session.

The interactive session starts with the lemma on the top of the goal stack. The Line 1 simply applies the automatic reasoner `FULL_SIMP_TAC` with a set of theorems, definitions and axioms that are included in the simplification set `srw_ss()` of the HOL4. We give as input to the `FULL_SIMP_TAC` also the lemma's definition: `lemma_w_xor_ex_property_def` in order to unfold its definition in the top goal. The `FULL_SIMP_TAC` is a tactic that simplifies the goal on top of the stack applying

the logical rules defined in the simplification set `srw_ss()`. With Line 2 the tactic `GEN_TAC` is repeatedly applied to instantiate all the universally quantified variables of the lemma's definition. The instantiation is done with arbitrary variables. With Line 3, the application of the tactic `STRIP_TAC` allows to move all the hypotheses of the lemma's definition to the assumption list of the goal stack. The Line 4 instantiates the universally quantified variable `phy_page` of the lemma's definition. With the Line 5 we assume an available theorem `pgrefs_thm` initializing all its universally quantified variables by the current system state, and the current physical block `phy_page`. The Line 9 applies a full simplification to the goal and to the assumptions. This simplification takes as input the definition of the reference counters `invariant_ref_cnt_def` in order to unfold this definition in the top goal an in the assumption list. The Line 10 specializes the definition of the reference counter with the current physical block `phy_page`. With Line 11 the automatic reasoner `RW_TAC` is applied, this is a tactic that rewrites the goal and the assumptions using the concepts of the simplification set `srw_ss()`. The application of this tactic generates 2 sub-goals. The first consists in proving that the executable reference counter is zero in case the physical block `phy_page` is writable. The second goal consists in proving that the writable reference counter is zero in case the physical block `phy_page` is executable. The two sub-goals are solved using a personalized tactic `count_refs_is_zero`. With this tactic we are able to show that, under the invariant assumption, it is impossible to find an entry of a page table granting an access to `phy_page` that is executable (for the first sub-goal) or writable (for the second sub-goal).

Chapter 5

Conclusions

In this work a possible application of the runtime monitoring technique has been analysed in the context of the Prosper project. The runtime monitor is considered as a security module with the goal of ensuring a consistent state of the system running on top of the Prosper hypervisor.

The first part of this work consists in studying the general design choices for the Prosper monitor. This is done based on the concept of *reference monitor*. In order to ensure a consistent system state, the Prosper monitor must provide a validation mechanism that satisfies the requirements of the *complete mediation*, the *tamper-resitance*, and the *verifiability*. The design choices adopted exploit the benefits of the virtualization to satisfy the tamper-resistance requirement. In fact, we consider that the runtime monitor is a guest of the Prosper hypervisor. In order to fulfil the complete mediation requirement, an interaction protocol between the hypervisor and the monitor has been established. The protocol states that, for each hypercall that is invoked by a guest of the hypervisor, this must be validated by the runtime monitor. Finally, in order to satisfy the verifiability requirement, we made the choice of identifying small and minimal checks for the validation mechanism.

The second part of this work consists in analysing, in general, what kind of security properties can be enforced by the Prosper monitor. This is done taking into analysis the hypercalls that are provided by the Prosper hypervisor to the guests. Since a hypercall is invoked for each operation that changes the memory layout of a guest, we conclude that the Prosper monitor is able to ensure a property, with an appropriate validation mechanism, only if the property constrains the

system memory layout. Furthermore, the property must be a safety property. This is concluded based on a general characterization of the enforceable properties by the class of Execution Monitors, that includes also reference monitors.

The third part studies a possible application for the Prosper monitor. This is done considering a system running on top of the Prosper hypervisor that is composed by the Linux kernel and the runtime monitor. We identified a safety property, denoted as $SAFE$, to be enforced by the monitor in order to protect the Linux kernel from malicious attacks of type code-injection. Through an appropriate analysis, and adopting a formal methodology, we showed that an additional property (denoted as $W \oplus X$) needs to be enforced. This property is known in the literature as *the executable space protection*, and it is necessary to maintain a consistent system state with the property $SAFE$.

An important part of this work is the identification of an appropriate validation mechanism that is able to ensure the properties $SAFE$ and $W \oplus X$. To this purpose, first we identified the *security-sensitive* operations for these properties. Then, we formalized sufficient conditions that are used to validate each security-sensitive operation. The last part of this thesis sketches a formal proof of the validation mechanism's correctness that has been fully verified using the proof assistant HOL4. Thanks to this formal verification, we demonstrate that the validation mechanism identified is able to ensure $SAFE$ and $W \oplus X$ over all the system states.

5.1 Related works

In this section we refer to two works from the literature that are related to the application of the Prosper monitor reported in this thesis. The first one proposes a framework, that is hypervisor based, to avoid malicious code injection attacks. This work differs from the Prosper monitor since it does not provide any formal verification of correctness. The second work enforces the property of the executable space protection to improve the Linux kernel security. This work differs from ours since the virtualization technology is not exploited.

Patagonix [24] Patagonix is a hypervisor based security module that detects and identifies executing binaries in order to avoid malicious code injection. The

Patagonix module depends only on the processor hardware to detect code execution and on the binary format specifications of executables to identify code and verify code modifications. Similarly to our work, the Patagonix security module relies on the *executable space protection property* that must be enforced by the hypervisor on which Patagonix is based. In [24] the authors present and discuss the Patagonix framework that is implemented as a prototype on the Xen 3.0.3 hypervisor [7].

Improving Linux kernel protection [23] This work addresses the problem of malicious code injection attacks in a Linux based system. Also in this work the authors rely on the enforcement of *the executable space protection* property to protect the Linux kernel from this kind of attack. In [23] an analysis of the Linux kernel memory management is presented along with an abstract model of this management. Therefore, the model checking methodology is applied to the model in order to verify whether the executable space protection property is satisfied. Since the analysis's results show that the Linux kernel does not guarantee this property, the authors propose some improvements to the Linux kernel.

5.2 Future works

The main future work for this thesis is the real implementation of the Prosper runtime monitor for protecting the Linux kernel from code injection. The main issue to be addressed in this sense is that the Linux kernel guest might not respect the property $W \oplus X$. All the Linux kernel's requests of memory configuration that make a physical block both writable and executable are not validated by the monitor. Therefore, under this security property the Linux kernel guest might not be able to execute. In our discussion and based on the top level specification proposed in Section 3.3.1, the Linux kernel would execute the same request whenever this is not validated by the runtime monitor, causing an infinite loop. In fact, the inference rule *insecurepriv* of the transition system proposed states that the handler ξ is simply the identity. To address this problem, one possibility could be the addition of an emulation layer to the architecture. This emulator implements the function ξ and it is invoked whenever a *prefetch abort* or *data abort* interrupts are raised by the Linux kernel. The emulator must find an appropriate strategy that

allows the Linux kernel execution without by-passing the monitor intermediation and control.

Bibliography

[1] HOL4 web page. `http://hol.sourceforge.net/`.

[2] Introduction to code signing. `http://msdn.microsoft.com/en-us/library/ie/ms537361(v=vs.85).aspx`.

[3] Provably secure execution platforms for embedded systems (Prosper) web page. `http://prosper.sics.se/`.

[4] The Xen project wiki web page. `http://wiki.xen.org/wiki/Main_Page`.

[5] Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[6] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[7] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003.

[9] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses

and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[10] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 223–234. ACM, 2013.

[11] Mads Dam, Roberto Guanciale, and Hamed Nemati. Machine code verification of a tiny ARM hypervisor. In Ahmad-Reza Sadeghi, Frederik Armknecht, and Jean-Pierre Seifert, editors, *TrustED'13, Proceedings of the 2013 ACM Workshop on Trustworthy Embedded Devices, Co-located with CCS 2013, November 4, 2013, Berlin, Germany*, pages 3–12. ACM, 2013.

[12] Mads Dam, Roberto Guanciale, and Hamed Nemati. Trustworthy virtualization of the ARMv7 memory subsystem. *To appear in SOFSEM 2015*, 2015.

[13] Victor Do. *Security Services on an Optimized Thin Hypervisor for Embedded Systems.* Masters thesis, Faculty of Engineering LTH at Lund University, 2011.

[14] Heradon Douglas. *Thin Hypervisor-Based Security Architectures for Embedded Platform.* Masters thesis, The Royal Institute of Technology, Stockholm, Sweden, 2011.

[15] Úlfar Erlingsson. The inlined reference monitor approach to security policy enforcement. Technical report, Cornell University, 2003.

[16] Anthony Fox and Magnus O Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving*, pages 243–258. Springer, 2010.

[17] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, New York, NY, USA, 1993.

[18] John Harrison. HOL light: A tutorial introduction. In *Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

[19] Trent Jaeger. Reference monitor. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 1038–1040. Springer, 2011.

[20] Narges Khakpour, Oliver Schwarz, and Mads Dam. Machine assisted proof of ARMv7 instruction level isolation properties. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2013.

[21] Ekkart Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

[22] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, and Srivaths Moderator-Ravi. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760. ACM, 2004.

[23] Siarhei Liakh, Michael Grace, and Xuxian Jiang. Analyzing and improving linux kernel memory protection: a model checking approach. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 271–280. ACM, 2010.

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

[25] T Ormandy and J Tinnes. Linux null pointer dereference due to incorrect proto ops initializations, 2009.

[26] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *SIGOPS Oper. Syst. Rev.*, 7(4):121–, January 1973.

[27] Leonid Ryzhyk. The ARM architecture. *University of New South Wales*, 2006.

[28] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[29] David Seal. *ARM architecture reference manual*. Pearson Education, 2001.

[30] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[31] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[32] D.E. Williams. *Virtualization with Xen(tm): Including XenEnterprise, XenServer, and XenExpress: Including XenEnterprise, XenServer, and XenExpress*. Elsevier Science, 2007.

[33] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, 2002.