PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Doctorado
en Informática

Formally verified countermeasures

Against cache based attacks in

Virtualization platforms

Juan Campo

2016

Formally verified countermeasures
Againsat cache based attacks in
Virtualization platforms
ISSN 0797-6410
Tesis de Doctorado en Informática
Reporte Técnico RT 16-02
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, 2016

PEDECIBA INFORMÁTICA

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

TESIS DE DOCTORADO EN INFORMÁTICA

Formally Verified Countermeasures Against Cache Based Attacks in Virtualization Platforms

Juan Campo
jdcampo@fing.edu.uy

October 2015

Thesis advisors: Gilles Barthe[1] and Gustavo Betarte[2]

[1] IMDEA Software
gilles.barthe@imdea.org

[2] InCo, Facultad de Ingeniería, Universidad de la República
gustun@fing.edu.uy

Formally Verified Countermeasures Against Cache Based Attacks in Virtualization Platforms

Abstract

Cache based attacks are a class of side-channel attacks that are particularly effective in virtualized or cloud-based environments, where they have been used to recover secret keys from cryptographic implementations. One common approach to thwart cache-based attacks is to use *constant-time* implementations, which do not branch on secrets and do not perform memory accesses that depend on secrets. However, there is no rigorous proof that constant-time implementations are protected against concurrent cache attacks in virtualization platforms; moreover, many prominent implementations are not constant-time. An alternative approach is to rely on system-level mechanisms. One recent such mechanism is *stealth memory*, which provisions a small amount of private cache for programs to carry potentially leaking computations securely. We weaken the definition of constant-time, introducing a new program classification called *S-constant-time*, that captures the behavior of programs that correctly use stealth memory. This new definition encompasses some widely used cryptographic implementations. However, there was no rigorous analysis of stealth memory and S-constant-time, and no tool support for checking if applications are S-constant-time.

In this thesis, we propose a new information-flow analysis that checks if an x86 application executes in constant-time or S-constant-time. Moreover, we prove that (S-)constant-time programs do not leak confidential information through the cache to other operating systems executing concurrently on virtualization platforms. The soundness proofs are based on new theorems of independent interest, including isolation theorems for virtualization platforms, and proofs that (S-)constant-time implementations are non-interfering with respect to a strict information flow policy which disallows that control flow and memory accesses depend on secrets. We formalize our results using the Coq proof assistant and we demonstrate the effectiveness of our analyses on cryptographic implementations, including PolarSSL AES, DES and RC4, SHA256 and Salsa20.

Keywords: Non-interference, cache-based attacks, constant-time cryptography, stealth memory, Coq

Formally Verified Countermeasures Against Cache Based Attacks in Virtualization Platforms

Resumen

Los ataques basados en el *cache* son una clase de ataques de canal lateral (*side-channel*) particularmente efectivos en entornos virtualizados o basados en la nube, donde han sido usados para recuperar claves secretas de implementaciones criptográficas. Un enfoque común para frustrar los ataques basados en cache es usar implementaciones de tiempo constante (*constant-time*), las cuales no tienen bifurcaciones basadas en secretos, y no realizan accesos a memoria que dependan de secretos. Sin embargo, no existe una prueba rigurosa de que las implementaciones de tiempo constante están protegidas de ataques concurrentes de cache en plataformas de virtualización. Además, muchas implementaciones populares no son de tiempo constante. Un enfoque alternativo es utilizar mecanismos a nivel del sistema. Uno de los más recientes de estos es *stealth memory*, que provee una pequeña cantidad de cache privado a los programas para que puedan llevar a cabo de manera segura computaciones que potencialmente filtran información. En este trabajo se debilita la definición de tiempo constante, introduciendo una nueva clasificación de programas llamada *S-constant-time*, que captura el comportamiento de programas que hacen un uso correcto de *stealth memory*. Esta nueva definición abarca implementaciones criptográficas ampliamente utilizadas. Sin embargo, hasta el momento no había un análisis riguroso de *stealth memory* y *S-constant-time*, y ningún soporte de herramientas que permitan verificar si una aplicación es *S-constant-time*.

En esta tesis, proponemos un nuevo análisis de flujo de información que verifica si una aplicación x86 ejecuta en *constant-time* o *S-constant-time*. Además, probamos que los programas (*S-*)*constant-time* no filtran información confidencial a través del cache a otros sistemas operativos ejecutando concurrentemente en plataformas de virtualización. La pruebas de corrección están basadas en propiedades que incluyen teoremas, de interés en sí mismos, de aislamiento para plataformas de virtualización y pruebas de que las implementaciones (*S-*)*constant-time* son no interferentes con respecto a una política estricta de flujo de información que no permite que el control de flujo y los accesos a memoria dependan de secretos. Formalizamos nuestros resultados utilizando el asistente de pruebas Coq, y mostramos la efectividad de nuestros análisis en implementaciones criptográficas que incluyen PolarSSL AES, DES y RC4, SHA256 y Salsa20.

Palabras clave: No interferencia, ataques basados en cache, criptografía constant-time, stealth memory, Coq

Acknowledgments

I would like to thank first my thesis advisors, Gilles Barthe and Gustavo Betarte, for their knowledge, motivation and support. Working with them has been an exciting and rewarding experience.

I also want to thank the evaluation committee: Joshua Guttman, Gerwin Klein, Maximiliano Cristiá, Mads Dam and Álvaro Tasistro, for their feedback and corrections to this document.

Many people made substantial contributions to the research presented in this thesis. Mauricio Chimento and Julio Pérez helped with some of the Coq proofs. I spent many hours discussing and developing much of the Coq formalization with Carlos Luna. It was a pleasure to work with him, and I hope we can continue doing research together for many more years. David Pichardie implemented the final version of the static analysis in CompCert presented in this work. His knowledge and expertise on the compiler and the techniques we used were instrumental in achieving the results we sought.

The time I spent at IMDEA Software working with Gilles was incredibly productive and motivating. I thank the people here and there that made that possible.

I am grateful for the opportunity and support provided by PEDECIBA Informática and in particular for letting me do the PhD without doing the Master's degree first. Javier Baliosian, Alberto Pardo and Alfredo Viola participated in the committee that allowed this exception. Thank you all for believing in me and my work.

The Instituto de Computación of Facultad de Ingeniería was also a big help during these years. I am very thankful to my colleagues at the courses I teach (Lógica and FSI), for working more than their share to allow me to focus on my research.

The time I spent doing research for this thesis has had a big impact on my day job as a sysadmin. I will be forever indebted to the guys at the Unidad de Recursos Informáticos, and in particular to its chief Jorge Sotuyo, for their support and for giving me the time to do this. I hope I can make up for it somehow in the future.

Finally, I would like to dedicate this thesis to Leticia. Without her patience and support over the years I would have never been able to see this through.

This work was partially funded by: i) VirtualCert: Towards a Certified Virtualization Platform (ANII-Clemente Estable, PR-FCE-2009-1-2568, Uruguay, 2010-2012); and ii) VirtualCert: Towards a Certified Virtualization Platform - Phase II (UDELAR-CSIC I+D, Uruguay, 2013-2015).

Contents

List of Figures

Chapter 1

Introduction

This thesis presents a formal approach for securely deploying and executing code that manipulates confidential information (such as user credentials or cryptographic keys) on a computing environment. Our main goal is to guarantee that this confidential information cannot be obtained by (does not *leak* to) an unauthorized user. One of the most common mechanisms to provide such a computing environment, where users are *isolated* from one another, is *virtualization*. We will see, however, that this mechanism is only a partial solution, due to the presence of *side-channels* via shared physical resources.

1.1 Problem statement

Virtualization platforms, which are the focus of our study, allow several operating systems (called *guest operating systems*) to coexist on commodity hardware, and provide support for multiple applications to run seamlessly on the operating systems running on top of them. Roughly, virtualization can be defined as the addition of a software layer, the virtual machine monitor (VMM) or hypervisor, between the hardware and the existing software that exports an interface (virtual machine or VM) at the same level as the underlying hardware. Figure 1.1 shows a diagram of a virtualization platform.

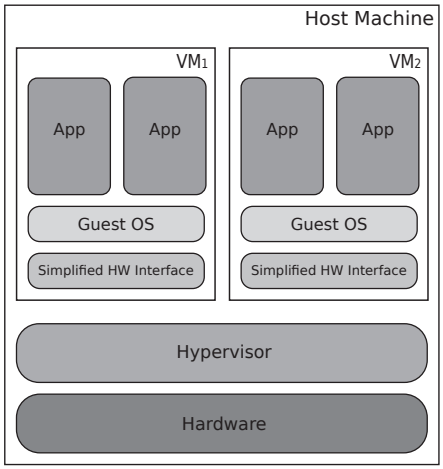

Figure 1.1: Diagram of a virtualization platform

Virtualization technologies first appeared in the late sixties as a way to efficiently share the expensive mainframe hardware of the time between a large group of users. After a period of obscurity in the nineties, it has become pervasive in high-integrity, safety-critical components in

embedded systems (mobile phone, airplanes), as well as in data centers and cloud computing infrastructures [110]. Historically there have been two main styles of virtualization: *full virtualization* and *paravirtualization*. In the first, each virtual machine is an exact duplicate of the underlying hardware, and the VMM provides no functionality except multiplexing the hardware resources (such as CPU, system memory, I/O devices) among multiple VMs [65]. This makes it possible to run unmodified operating systems on top of it. When an attempt to execute a privileged instruction by the OS is detected, the hardware raises a trap that is captured by the hypervisor and then it emulates the instruction behavior. In the paravirtualization approach, each virtual machine is a simplified version of the physical architecture. The guest operating systems must be modified to perform calls to the hypervisor (called *hypercalls*) instead of directly accessing the hardware. A hypercall interface allows OSs to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. An example use of a hypercall is to request a set of page table updates, in which the hypervisor validates and applies a list of updates, returning control to the calling OS when this is completed.

One of the most important features of a virtualization platform is to guarantee that applications with different security policies can execute securely in parallel. This is accomplished by ensuring that the guest operating systems run **isolated** from one another, i.e. they have independent behavior and cannot learn about or influence each other's information and actions. In order to guarantee isolation and to keep control of the platform, a hypervisor makes use of the different execution modes of the CPU. Traditionally[1], the hypervisor itself runs in *supervisor mode*, in which all CPU instructions are available; while the guest operating systems run in *user mode* in which privileged instructions cannot be executed.

The increasingly important role of virtualization in software systems makes it a prime target for formal verification. Indeed, the complexity and critical nature of these platforms make faults both more likely and severe. Formal methods allow studying, understanding and guaranteeing the correctness of the behavior of such systems. In [81], Klein provides a very complete survey of software verification, with a special emphasis on verifying operating systems. Recently, several projects have set out to formally verify the correctness of operating system kernel and hypervisor implementations. The most prominent initiatives are the Microsoft Hyper-V verification project [47, 92], and the L4.verified project [82]. These works are discussed in some further detail in Chapter 7.

Reasoning about implementations provides the ultimate guarantee that deployed hypervisors provide the expected properties. There are however significant hurdles with this approach, especially if one focuses on proving security properties rather than functional correctness. The complexity of formally proving non-trivial properties of implementations might be overwhelming in terms of the effort it requires; worse, the technology for verifying some classes of security properties may be underdeveloped: specifically, there is currently no established method for verifying security properties involving two system executions, a.k.a. 2-properties [46], for implementations. In particular, the main results of this thesis are non-interference properties, which cannot be expressed in terms of only one system execution. In addition, many implementation details are orthogonal to the security properties to be established, and may complicate reasoning without improving the understanding of the essential conditions that guarantee isolation among guest operating systems. Thus, there is a need for complementary approaches where verification is performed on idealized models that abstract away from the specifics of any particular hypervisor,

[1]Some modern processors now use a hierarchy of execution modes. Specifically, with the increased popularity of virtualization technologies, new modes have been added in order to more easily support hardware assisted virtualization. Examples include ARM virtualization extensions [123], Intel Vanderpool [122], AMD Pacifica [14], and more recently, Intel SGX [98].

and yet provide a realistic setting in which to explore the security issues that pertain to the realm of hypervisors.

Despite ongoing efforts, showing that virtualization platforms ensure isolation remains a significant challenge. Problems arise due to the sharing of hardware resources between the different VMs running concurrently. Not only does the hypervisor need to carefully control the sharing of resources and validate the guest operating systems behavior, making it a very complex piece of software; but there can be **side-channels** that can be exploited to bypass the isolation mechanisms imposed by the hypervisor. Side-channels attacks obtain secret information from a victim system by observing the execution of the system on a physical environment, and analyzing measurable differences on the shared hardware that are related to its secrets.

The cache is one such shared hardware resource. It is a fast storage device used by CPUs to store a copy of data from main memory to reduce the overhead necessary to access memory data. When the processor needs to read a memory value, it fetches the value from the cache if it is present (*cache hit*), or from memory if it is not (*cache miss*).

Cache-based attacks are a very powerful attack vector that use the cache as a side-channel, allowing a malicious party to obtain confidential data through observing timing differences in program execution. They are widely applicable, but are specially devastating against cryptographic implementations that form the security backbone of many Internet protocols (e.g. TLS) or wireless protocols (e.g. WPA2). Known targets of cache-based attacks include widely used implementations of DES [121], AES [31, 120] and RC4 [42]. These attacks have been constantly improved and applied in different settings; see for example [74, 75, 129]. In Chapter 7 we describe these and other works in further detail.

Cache-based attacks may be sophisticated, but their underlying idea is relatively simple: an attacker observes the cache to gain partial information about the sequence of reads and writes performed by a program during execution; using this knowledge, he successfully retrieves confidential data of the program.

In practice, one distinguishes between three types of attacks: trace-driven, time-driven, and access-driven [5]. In a trace-driven attack, the adversary has the ability to observe if each memory access during the victim program execution generated a cache miss or a cache hit. Time-driven attacks are less restrictive since the adversary can only measure an aggregate value related to the total number of cache hits and misses (for example, the total execution time of the victim). Finally, in an access-driven attack, the adversary has visibility over the cache line sets that the victim accesses during its computation.

These types of attacks have also been successful in cloud computing environments [109]. In such settings, a malicious tenant can manage that an operating system under its control co-resides with the operating system which executes the program targeted by the attacker. This allows the attacker to share the cache with its victim and therefore to exploit the possible side-channels that arise, in order to retrieve confidential information from the victim program.

Several countermeasures have been proposed for these attacks, see for example [106, 126, 88, 48, 120, 53]. One approach is to develop applications that do not leak information through the cache, such as making implementations **constant-time**, i.e. programs that do not branch on secrets and do not perform memory accesses that depend on secrets. There are some cryptographic algorithms that are designed to be constant-time, such as SHA256, TEA, Salsa20; and some that can be made constant-time by modifying their implementations, for example, AES [77], DES, RC4, and even RSA. There are even some general techniques for turning implementations of cryptographic algorithms constant-time [34]. However, there is no rigorous proof that constant-time algorithms are indeed protected against cache-based attacks when executed concurrently on virtualization platforms with shared cache. Moreover, many common cryptographic implementations such as PolarSSL AES, DES, and RC4 do make array accesses that depend on secret keys for efficiency

reasons and are therefore not constant-time.

A different and more permissive approach is to allow implementations that are not constant-time, but to deploy system-level countermeasures that prevent an attacker from drawing useful observations from the cache. Some of these mechanisms are transparent to applications, but sacrifice performance: instances include flushing the cache at each context switch [120] or randomizing its layout [125]. We formally analyzed the first countermeasure in a previous work, see Section 1.3.

Other mechanisms are not transparent and must be used correctly, either via APIs or via compilers that enforce their correct usage. For example, *warming* [107] loads all tables that may be accessed with secret-dependent indices into the cache immediately after a context switch. These countermeasures are generally applicable, but cache size and decreased performance might be an issue.

One lightweight mechanism is **stealth memory** [57]; in contrast to many of its competitors, stealth memory can be implemented in software, does not require any specific hardware and does not incur a significant performance overhead. Informally, stealth memory enforces a locking mechanism on a small set of cache lines, called *stealth cache lines*, saves them into protected memory and restores them upon context switches, thereby ensuring that entries stored in stealth cache lines are never evicted, and do not leak information. From an abstract perspective, memory accesses to stealth addresses, i.e addresses that map to stealth cache lines, become "hidden" and have no visible effect. Thus, applications can perform memory accesses that depend on secrets without revealing confidential information, provided these accesses are done on stealth addresses. From an application point of view, stealth memory provides a convenient means to protect some actions during program execution, and allows making programs secure against cache-based attacks. A recent implementation and evaluation of stealth memory [80] shows how this effect can be achieved with a modest amount of stealth memory, and a minimal performance overhead.

Although early work on stealth memory suggests that several prominent cryptographic implementations can be made secure by using this technology correctly, this class of programs has not been formally studied before, and in particular, there is no rigorous security analysis of stealth memory and no mechanism to ensure that application code uses it correctly.

1.2 Contributions

This work focuses on the study of constant-time implementations; giving formal guarantees that such implementations are protected against cache-based attacks in virtualized platforms where their supporting operating system executes concurrently with other, potentially malicious, operating systems. We also study stealth memory as another flexible and efficient countermeasure against these attacks. Moreover, we provide support for deploying constant-time and applications using stealth memory, in the form of type-based enforcement mechanisms on x86 implementations. The mechanisms are integrated into CompCert, a realistic verified compiler for C [93]. Additionally, we experimentally validate our approach on a set of prominent cryptographic implementations. To achieve these goals, we make the following contributions:

- We develop a model of virtualization that accounts for virtual addresses, physical and machine addresses, memory mappings, page tables, translation lookaside buffer (TLB), and cache; and provides an operational semantics for a representative set of actions, including reads and writes, allocation and deallocation, context switching, and hypercalls. We prove a non-interference result on the model that shows that an adversary cannot discover secret information using cache side-channels, from a constant-time victim. This model is presented

in detail in Chapter 2, and the non-interference result in Chapter 3.

- We define an analysis for checking if x86 applications are constant-time. Our analysis is based on a type system that simultaneously tracks aliasing and information flow. For convenience, we package our analysis as a certifying compiler for CompCert. Our certifying compiler takes as input a C program whose confidential data is tagged with an annotation High, and checks whether it is constant-time. We prove that constant-time programs are non-interfering with respect to an information flow policy which mandates that the control flow and the sequence of memory accesses during program execution do not depend on secrets. The policy is captured using an operational semantics of x86 programs where transitions are labelled with their read and write effects. These results are presented in Chapter 4.

- We provide a formal proof that constant-time programs are protected against cache-based attacks in virtualization platforms. The proof contemplates a very strong threat model with a malicious operating system that controls the scheduler, executes concurrently with the operating system on which the victim application runs, and can observe how the cache evolves throughout the execution. The theorem and a sketch of the proof are presented in Section 4.7.

- We extend our virtualization model with stealth memory primitives, and prove the isolation result in this new setting. We also introduce a new program classification, which we call *S-constant-time*, that captures the behavior of programs that correctly use stealth memory. As a significant contribution of these extensions, we obtain the first rigorous security analysis of stealth memory. These results are presented in Chapter 5.

- We successfully evaluate the effectiveness of our framework on several cryptographic implementations, including AES, DES, and RC4 from the PolarSSL library, and SHA256, Salsa20. We observe that these implementations are constant-time or S-constant-time. By measuring their stealth memory requirements, we conclude that S-constant-time implementations can be executed on virtualization platforms with a small amount of stealth memory, and hence with a very limited overhead (the performance overhead of using stealth memory is proportional to its size). We present this evaluation in Chapter 6.

- We formalized and machine checked most of our results in the Coq proof assistant (over 50,000 lines of code). The formalization[2]. is based on the first formal model of stealth memory. The model is a significant development in itself (over 10,000 lines of code) and is of independent interest.

  The only result presented in this thesis that was not completely mechanized in Coq (as of this writing) was the valid state invariance for the basic model (without stealth memory). This property (Lemma 2.4.1) was already proved for other versions of the model [20, 21]. While long and laborious, this proof is conceptually easy, and we therefore decided not to redo it and leave it for future work. We have proved, however, the invariance of four new stealth conditions (presented in Section 5.1.1), and also proved that the semantics of the new stealth memory actions (presented in Section 5.1.2) keep all state validity conditions invariant.

[2]The formal development is available at `http://www.fing.edu.uy/inco/grupos/gsi/documentos/tesis/coq-devel.tar.gz`, and can be verified using Coq , version `8.4pl6`

1.3 Summary of publications

This thesis builds upon and extends a number of previously published papers:

- G. Barthe, G. Betarte, J. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In FM 2011, pages 231–245. Springer-Verlag, 2011.
- G. Barthe, G. Betarte, J. Campo, and C. Luna. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In CSF 2012, pages 186–197, 2012.
- G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, and C. Luna. Formally verified implementation of an idealized model of virtualization. In R. Matthes and A. Schubert, editors, 19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France, volume 26 of LIPIcs, pages 45–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informative, 2013.
- G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, pages 1267–1279, New York, NY, USA, 2014. ACM.

In each paper I worked on the analysis and development of the different formal specifications in Coq and the formulation and proof of the relevant security properties, in particular the isolation results. In addition, I was involved in the design of the static analysis for constant-time and S-constant-time programs, but not on the final implementation in CompCert. I also participated in the integration of the static analysis and the idealized virtualization model, in order to state and prove an isolation theorem that relates both models, see Chapters 3 and 4 for more details. I have also collaborated in the analysis of results and writing of those papers.

In this thesis we will focus on and expand the results presented in the last work [22]. The others, since they are the basis on which this work is built, and provide interesting complementary approaches to the ones followed here, will be briefly described in this section.

In [20], we developed a minimalistic model of a hypervisor, and formally proved that the hypervisor correctly enforces isolation between guest operating systems. We also showed that under mild hypotheses, it guarantees basic availability properties to guest operating systems. In this first model we abstracted away many specifics of memory management such as caches and TLBs. Instead, our model focused on the aspects that were most relevant for memory isolation properties. Specifically, we showed that an operating system can only read and modify memory it owns, and proved a non-influence property [103] stating that the behavior of an operating system is not influenced by other operating systems. In addition, our model allowed reasoning about availability; we proved, under reasonable conditions, that all requests of a guest operating system to the hypervisor are eventually attended, so that no guest operating system waits indefinitely for a pending request. Overall, our verification effort showed that the model was adequate to reason about safety properties (read and write isolation), 2-safety properties (OS isolation), and liveness properties (availability).

In [21] we continued our work by extending the model with a formalization of a VIVT cache and TLB. Then, drawing inspiration from physically observable cryptography [99], we considered an extended model of traces in which operating systems can draw observations on the history of the cache, therefore leading to a possible information leak through cache side-channels. The resulting model allowed us to reason about the class of synchronous access-driven cache-based attacks; in particular, we proved that flushing the cache upon switching between guest operating systems ensures OS isolation and prevents such attacks. The second main contribution of this

work was a machine-checked proof of what we called *transparency*. Transparency states that the virtualization platform is a correct abstraction of the underlying hardware, in the sense that a guest operating system cannot distinguish whether it executes alone or together with other systems. Transparency is a 2-safety property; its formulation involves an erasure function which removes all the components of the states that do not belong to some fixed operating system. We defined an appropriate erasure, established its fundamental properties, and derived the transparency property of the virtualization model.

While that work provided the basis to the extensions presented in this document, the main properties were somewhat simpler. The reason for this is that the previous model had a write-through cache policy. That allowed us to derive the equivalence relation for cache from the equivalence of memory (the set of cache values were always included in memory), and reuse the isolation proof presented in [20]. In the model presented here, the write policy is left abstract, so it is necessary to define the cache equivalence explicitly. More importantly, due to the flushing of the cache, no condition on the behavior of the victim was required, so there was no need to ensure that application code behaves correctly.

The semantics of the two previous works only considered correct action execution. In [23] we extended the semantics with error handling. We implemented a hypervisor in the programming language of Coq, and proved that it realizes the axiomatic semantics. Although it remained idealized and far from a realistic hypervisor, the implementation arguably provided a useful mechanism for validating the axiomatic semantics. This implementation was total, in the sense that it computed for every state and action a new state or an error. Thus, soundness was proved with respect to an extended axiomatic semantics in which transitions may lead to errors. The second contribution of this paper was a proof that OS isolation remained valid for executions that may trigger errors.

1.4 Outline

The remaining of the thesis is structured as follows. We present the virtualization model in Chapter 2. In Chapter 3 we describe in detail the adversary model used, and prove our main isolation result. Chapter 4 presents the static analysis for x86 applications. The extension of these results to stealth memory is presented in Chapter 5. The experimental results on cryptographic implementations are described in Chapter 6. Finally, in Chapter 7 we present the related work and Chapter 8 concludes the thesis.

Chapter 2

Virtualization Model

This chapter presents our idealized model of virtualization. This formal model focuses on the memory management of virtualization platforms, and it will allow us to reason about isolation properties, cache attacks and its countermeasures. Our modeling choices were originally guided by paravirtualization platforms, such as Xen [19], and specifically by Xen on ARM [73], and we will therefore use terminology from this technology.

Our model is a transition system with states and state transformers (called *actions*). These actions will be defined using axiomatic semantics, with pre and post-conditions. The states in the model will represent the memory structures of a machine running a hypervisor, and several operating system running on top of it. These structures include the main memory of the platform, various kinds of memory spaces, and a cache and TLB. In Section 2.1, we present a general overview of the virtualization model we have developed. Then, we describe the formal details of the model states and actions in Sections 2.2 and 2.3. Finally, in Section 2.4 we show that the actions preserve an appropriate valid state property that guarantees basic qualities of the model.

2.1 Model overview

Our model captures the main components of memory management in a virtualization platform. The memory and related resources, such as the cache and TLB, need to be securely shared between the guest operating systems.

In order to share the available memory between different application or processes, modern operating systems and hypervisors use a memory management mechanism called *virtual memory*. This mechanism provides each process with a unique memory space, and the processes access their own memory values using *virtual addresses* that are mapped to real addresses by a piece of hardware called the *memory management unit* (MMU). Hypervisors have the added task of providing the guest operating systems with a contiguous memory region for their correct operation. This is done by using an extra memory address space that maps addresses as seen by guest operating systems (called *physical addresses*) to the real memory addresses (called *machine addresses*).

The mapping between the virtual address space and machine memory is handled by the hypervisor and it is stored in special memory areas called page tables. There can be many levels of indirection in the page table structure (called multilevel page table) to use a more efficient representation of the memory space of a process. In this work we will use a single-level page table structure, represented by a page in memory, that stores the virtual to real address mapping. Each application of each guest operating system will have its own page table, and therefore its own virtual address space.

In addition to the page table structure, the TLB can be used to speed up the translation

from virtual to machine memory addresses. It is a class of cache: in the event of a TLB hit, the address corresponding to a given virtual address is obtained directly, instead of searching for it in the process page table (which resides in main memory).

One of the main components of our model is the cache. There are different ways to identify cache entries. The cache may potentially be accessed either with a physical address (physically indexed cache) or a virtual address (virtually indexed cache). There are, in turn, two variants of virtually indexed caches: those where the entries are tagged with the virtual address (virtually indexed, virtually tagged or VIVT cache) and those which are tagged with the corresponding physical address (virtually indexed, physically tagged or VIPT cache). We have studied both types of cache, see [21] and [22]; here we will use VIPT caches, as presented in the second work.

There are several alternative policies for implementing cache content management, in particular concerning the update and replacement of cache information. A replacement policy is one that specifies the criteria used to remove a value when the cache is full and a new value needs to be stored. Some common policies remove the least recently used (LRU), the most recently used (MRU) or the least frequently used value (LFU). We model an abstract replacement policy which can be refined to the ones most commonly used.

A write policy specifies how the modification of a cache entry impacts the memory values: a *write-through* policy, for instance, requires that values are written both in the cache and in the main memory. A *write-back* policy, on the other side, requires that values are only modified in the cache and marked dirty, and updates to main memory are performed when a dirty entry is removed from the cache. In the present work we again model an abstract write policy that can handle both strategies.

The type of cache we are modeling inherently has problems in the presence of aliases (two different virtual addresses that map to the same memory page), known as *the synonym problem*. In this case, the page can potentially be cached twice (once for each virtual address) in both VIVT and VIPT caches. If this happens, an update to one of the entries might leave the other one inconsistent. Standard solutions for this problem include specialized hardware that detects and handles aliases, flushing the cache, or detecting aliases and avoiding caching more than one writable page in the cache. We follow the last approach, and make aliased pages *non-cacheable*, so that there is never more than one copy of a memory page in the cache.

2.1.1 What we do not model

As is apparent from the previous description, the memory model we have designed is relatively complete. There are however, important components we have decided not to include in the model, since they are not directly related to the memory management behavior of the platform, or are somewhat orthogonal to the kind of analysis we plan to perform on the model. Including these components is however an interesting line of future research, and would make our model even more complete and realistic.

Related to the virtual memory management, there is a technique called *transparent page sharing*, that can be used in virtualization implementations (specifically in VMWare products) to reduce the amount of memory required by the virtual machines. This technique works by allocating only one copy of pages with the same contents between different operating systems (or different applications inside a given operating system). All guest operating systems would access this shared page (by mapping the virtual address of each OS to the same memory page). When the page is modified, the hypervisor must create a copy of it for the system that is doing the modification (this technique is called *copy-on-write*). Recently, some attacks that exploit transparent page sharing have been published [74]; and this led the vendor to disable the functionality [124]. Our model does not include this mechanism, and its study is left as future work.

Another aspect we have not included in the model is a different type of cache called *instruction cache*. In addition to the cache mentioned before (which can be called data cache), some architectures have an instruction cache to store fetched program instructions from memory and avoid a memory access for each instruction executed. This is specially efficient in loops, where the same sequence of instructions is used repeatedly. Even though instruction caches can also be exploited to leak information [6], we do not include this kind of caches in our model. Modeling it would require a much more detailed model of the processor and instructions (including modeling the fetch of the instructions from memory), and storing the program code in memory. Modeling program execution at this level of detail would require some work, but should be feasible, and it is left as future work.

We have also not considered input/output (IO) devices (such as hard disk drives, network interfaces, or input devices). These resources are shared between guest operating systems, and are therefore a possible cause of information leakage. One possible approach for modeling IO devices is to treat them as special memory regions. There are even cases where devices directly access main memory, and data is mapped from memory to the device (this technique is called direct memory access, or DMA).

Accesses to IO devices are possibly asynchronous, meaning that reading to one of these devices could block the process until the data is available; the device would then tell the operating system that the data is ready to be used; and finally the process would be unblocked to consume the data. This asynchronous access mechanism implies that it would be necessary to model an interruption mechanism, which would complicate our model somewhat. However, it would not be very different from the hypercall mechanism we have in our model (where writing to a page table, for example, essentially blocks the process until the hypervisor validates and executes the action on behalf of the calling operating system).

2.1.2 Variants and extensions

There are many possible variants and extensions that can be performed to our model. One possibility is to adapt it to other virtualization paradigms. While our model is based on paravirtualization, it can be easily adapted to more closely model a full-virtualization platform by treating hypercalls as traps, triggered when an operating system attempts to perform a privileged instruction (such as writing to a page table).

One important extension we have not undertaken yet is the concurrent execution of instructions by means of either multi-core or multithreading processors. Multi-core architectures provide more than one complete processing unit within the main CPU component, while multithreading allows using these processing units more efficiently by, for example, allowing several execution threads of a process to execute concurrently.

Formally modeling concurrency has its challenges. The main problem is to give a semantics for the model, which would have to be possibilistic and our isolation results would therefore also be possibilistic. Since this is not the kind of result we are aiming at, we focus on an execution model with only one process (and one guest operating system) executing at a given time, and leave the modeling of concurrent execution as future work.

2.2 States

In this section we give an overview of the state of the idealized virtualization model, which includes the main memory of the platform, various kinds of memory spaces, and the cache and TLB. In Figure 2.1 we show a high level diagram of the memory model.

The *machine memory* is the real machine memory. A mechanism of page classification was

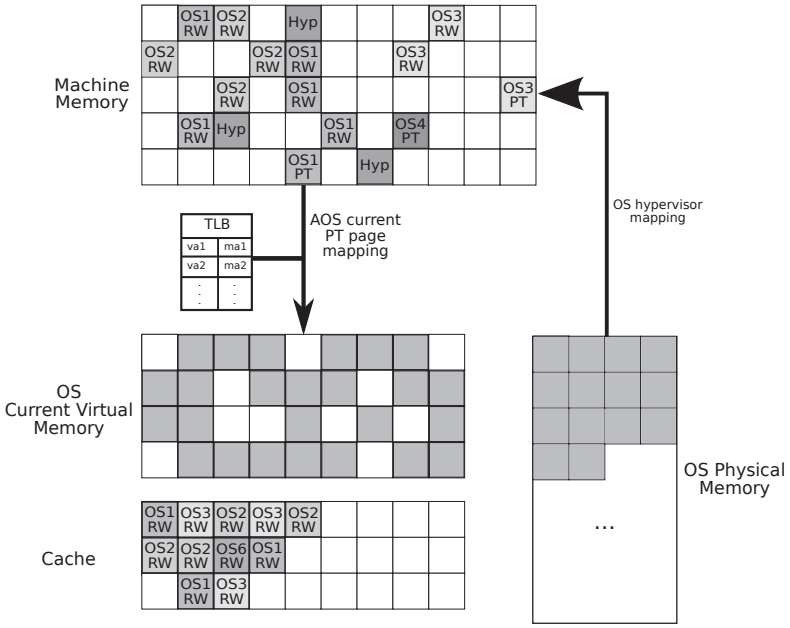

Figure 2.1: Memory model of the platform

introduced in order to cover concepts from virtualization platforms, in particular Xen [19]. The model considers that each machine address points to one memory page. Each page has at most one owner: a particular guest OS or the hypervisor. The pages are also classified as either a data page with read/write access or a page table, where the mappings between virtual and machine addresses reside. An OS is required to register and classify a page before being able to use it. The set of machine addresses (written Ma) model real hardware memory on the host machine and are never directly used by the guest OS.

In usual architectures with paging, the addresses are composed of two parts: a page identifier and an offset inside the page. We use an abstraction of this addressing mode, and do not distinguish between accesses at different offsets inside the page. All machine addresses will therefore refer to a different page, and we will only record the access to the memory page, not the specific data inside it. This gives us a good level of abstraction to reason about memory accesses, while still capturing the behavior of actual implementations.

The *physical memory* is an address space exposed by the hypervisor for the use of the guest OSs kernel. The physical addresses (whose type is written Pa) are provided by the hypervisor in order for the guest operating systems to use a contiguous memory space when referencing its own pages. The mapping between physical and machine addresses is managed exclusively by the hypervisor and it is transparent to the guest operating systems.

Virtual addresses (denoted Va) are used by applications running on guest operating systems. The hypervisor maintains page tables that map virtual addresses to machine addresses in special memory pages. The operating systems must call the hypervisor to modify these mappings.

Each OS has a designated portion of its virtual address space that is reserved for the hypervisor. This is done in order to avoid a context switch by the hypervisor to access its own memory, when an operating system makes a hypercall. This way, hypervisor data is always accessible through the virtual address space of all operating systems. We say that a virtual address va is *accessible* by the OS ($os\_accessible(va)$) if it belongs to the virtual address space of the OS which is not reserved for the hypervisor.

In Figure 2.2 we give the formalization of the platform state, which consists of a collection of components that we now proceed to describe.

Va, Pa, Ma		virtual, physical and machine addresses
OSId		OS identifier
HC	$:= new \mid del \mid lswitch \mid pin \mid unpin \mid \perp$	hyper calls
OSData	$:=$ Pa $\times$ HC	OS data
GuestOSs	$:=$ OSId $\mapsto$ OSData	guest OSs
OSActivity	$:= running \mid waiting$	OS activity state
EM	$:= usr \mid svc$	execution mode
ActiveOS	$:=$ OSId $\times$ OSActivity	active OS
PageContent	$:= RW(\text{Value}) \mid PT(Vadd \mapsto \text{Ma}) \mid \perp$	page content
PageOwner	$:= Hyp \mid OS(\text{OSId}) \mid \perp$	page owner
Page	$:=$ PageContent $\times$ PageOwner $\times$ Bool	memory page
Memory	$:=$ Ma $\mapsto$ Page	memory map
HyperMap	$:=$ OSId $\times$ Pa $\mapsto$ Ma	hypervisor map
CacheData	$:=$ Va $\times$ Ma $\mapsto_{K_c}$ Page	cache data
CacheIndex	$:=$ Va $\mapsto$ Index	cache index
CacheHistory	$:=$ Index $\mapsto$ Hist	cache history
Cache	$:=$ CacheData $\times$ CacheIndex $\times$ CacheHistory	VIPT cache
TLB	$:=$ Va $\mapsto_{K_t}$ Ma	TLB
SLST	$:=$ GuestOSs $\times$ ActiveOS $\times$ HyperMap $\times$ Memory $\times$ Cache $\times$ TLB	Platform state

Figure 2.2: Formal definition of the state

Operating systems

We define a type OSId of identifiers for guest operating systems. The state contains information about each guest OS current page table, which is a physical address, and information on whether it has a hypercall pending to be resolved. A hypercall is a privileged functionality exported by the hypervisor to the guest OSs. The list of availiable hypercalls is given by the type HC.

Formally this information is captured by the GuestOSs mapping that associates OS identifiers with objects of type OSData.

Active OS and execution modes

One of the operating systems is active at any given time (ActiveOS). For this OS, in addition to its identifier, we register whether it is currently running or waiting for a hypercall to be resolved. Active OS execution mode is formalized by the type OSActivity.

Most hardware architectures distinguish at least two execution modes, namely *user mode* (*usr*) and *supervisor mode* (*svc*). These modes are used as a protection mechanism, where *privileged* instructions are only allowed to be executed in supervisor mode. In our model, guest OSs execute in user mode while the hypervisor executes in supervisor mode. When a guest OS needs to execute a privileged operation, it requests the hypervisor to do it on its behalf. After requesting the hypervisor to execute some service, the active guest OS will turn to execution mode *waiting* until the service is completed and the execution control returned, switching then its execution mode to *running*. Execution modes are formalized by the type EM.

Mappings

The mapping that associates, for each OS, physical to machine addresses is managed by the hypervisor. This mapping, which is formalized as the component HyperMap of the state, might

be treated differently by each specific virtualization platform. There are platforms in which this mapping is public and the OS is allowed to use machine addresses. The physical-to-machine address mapping is modified by the actions `page_pin` and `page_unpin`, as described in Section 2.3.

The Memory is formalized as a mapping from machine addresses to pages. A memory page (Page), consists of page content (PageContent) and metadata. The content is either a readable/writable value, an OS page table (a mapping from virtual to machine addresses), or nothing (an unallocated or unused page).

Each OS has a collection of page tables associated (one for each application executing on the OS) that map virtual addresses into machine addresses. When executed, the applications use virtual addresses, therefore on context switches the current page table of the OS must change so that the currently executing application may be able to refer to its own address space. Page tables updates are done exclusively by the hypervisor (neither applications nor guest OSs have permission to do so); and every memory address accessed by an OS needs to be associated to a virtual address. The model must guarantee the correctness of those mappings, namely, that every machine address mapped in a page table of an OS is owned by it. We consider an abstract type of values equipped with an equality relation, and we assume given a distinguished value $\perp$ when the value is undefined. In particular we abstract away implementation details such as encoding, size, etc.

The page metadata contain a reference to the page owner, of type PageOwner, which can be either the hypervisor, an OS, or $\perp$ and a flag indicating whether the page can be cached or not.

Cache

The figure also shows the cache, which is accessed by a pair of virtual and machine addresses (modeling a VIPT cache) and holds a subset of the readable/writable memory pages. The size of the cache is bounded by a positive fixed constant K_c.

In VIPT caches indices of the cache are derived from the virtual addresses, but each entry is tagged with the machine address. This avoids the need of flushing the cache on every context switch, and therefore requires less software management.

The cache can be seen as a collection of data blocks or *cache lines* (which are pages in our model) that are accessed by cache indices. There is a mapping CacheIndex from virtual addresses to cache indices. Since caches are usually set associative, there are many virtual addresses that map to the same index. All data that is accessed using the same index is called a *cache line set*. As defined in [80] we assume that the *inertia* property holds for the cache. This property states that after adding an entry to the cache in a virtual address va and replacing some entry, the evicted address is in the same cache line set as va.

We also model historic access information as the abstract component of the cache CacheHistory. This is used by the replacement policy to decide the entry to be evicted from the cache. The replacement policy selects an entry to remove when a new entry is to be added and the corresponding cache line is full. There are various cache policies in use (LFU, LRU, PLRU, etc). We left the actual policy of the model abstract, and only require that it selects the same entry for removal from two caches, as long as their histories are the same.

This requirement is necessary in order to have a deterministic policy that consistently selects the same entry for removal. We will ultimately show that the same attacker entries are removed from the cache (implying that cache based attacks are impossible), see the discussion in Section 3.1. It is therefore necessary that the selection of the entry to remove is done based on specific public information (in our case, the cache history) that can be proved to be equal during two executions of the model.

Most replacement policies are consistent with our assumption. For example, the LRU policy (*least recently used*) would select the oldest entry from the cache line set for eviction. The history

in this case would store information on when each entry was accessed. When an entry is to be evicted, the one accessed least recently would be selected. In this case, if two caches have the same history and entries, and an eviction takes place, the policy will select the same entry in both cases. The LRU policy can therefore be shown to comply with the required assumption.

One important replacement policy that is problematic is *random replacement* (RR). With this policy, it would not be possible to prove that the attacker will always see the same entries evicted (since the selection is completely random). One way to handle this is to consider a *pseudo-random number generator* (PRNG). A PRNG generates a deterministic approximation of a random sequence, starting from an initial number called the *seed*. In this case we would store the seed and the number of calls to the PRNG in the cache history, so that given two caches with the same histories and entries, the replacement policy will select the same entry for eviction.

TLB

The TLB is used in conjunction with the current page table of the active OS to speed up translation of virtual to machine addresses. The TLB is flushed on context switches and updates are done simultaneously in the page table, so its management is simpler than the cache. Therefore we do not record the TLB access history, as it is not necessary to write back evicted TLB entries. The size of the TLB is bounded by a positive fixed constant K_t.

2.2.1 Access functions

In this section we define the helper functions used to predicate and manipulate the components of the state. We will give the type of each function or predicate, and an high level explanation of its behavior.

$os\_accessible(va : \mathsf{Va}) : Prop$ is a predicate indicating whether a virtual address is accessible to an OS or if it is reserved for the hypervisor.

$get\_page\_mem(s : \mathsf{SLST}, va : \mathsf{Va}) : Option(\mathsf{Ma} \times \mathsf{Page})$ returns the machine address ma and the page pg mapped from the virtual address va in the active OS current page table mapping. This function first searches for va in the TLB, and if it is not there, in the current page table in memory, to find the ma. Once it has the ma, it looks for an entry indexed with (va, ma) in the cache. It returns the corresponding page in the cache, or the page in ma from main memory if it is not cached.

$[\ ]_{pol} : \mathsf{Memory} \to \mathsf{Ma} \to \mathsf{Page} \to \mathsf{Memory}$ similar to a regular memory mapping update, but depends on the selected write policy. It either writes the new page in memory (in write-through policy), or leaves the memory unchanged (write-back policy). The application of this function will be written: $mem[ma := pg]_{pol} = mem'$.

$cache\_add(c : \mathsf{Cache}, va : \mathsf{Va}, ma : \mathsf{Ma}, pg : \mathsf{Page}) : \mathsf{Cache} \times Option\ (\mathsf{Ma} \times \mathsf{Page})$ adds entries in the cache. It returns the new cache and an optional entry that was selected for replacement and removed from the cache. This function is parameterized by an abstract replacement policy that determines which elements are evicted from a full cache line, and guarantees that the *inertia* property holds for the cache.

$get\_page\_va\_ma(s : \mathsf{SLST}, va : \mathsf{Va}, ma : \mathsf{Ma}) : Option\ (\mathsf{Page})$ returns the page corresponding to the cache entry (va, ma), if it exists; otherwise returns the page in ma from memory.

$get\_page\_hyp(s : \mathsf{SLST}, os : os\_ident, pa : \mathsf{Pa}) : Option\ (\mathsf{Ma} \times \mathsf{Page})$ returns the machine address ma and the page pg in memory, mapped from the physical address pa in the hypervisor mapping for the OS.

$memory\_alias : (mem : \mathsf{Memory}, va : \mathsf{Va}, ma : \mathsf{Ma}) : Prop$ is a predicate that indicates whether a page table maps a virtual address other than va to ma. This is used to guarantee that no aliased pages are cached, since this would be incorrect for a write-back cache policy.

$remove\_cache\_va(c : \mathsf{Cache}, cpt : \mathsf{Page}, va : \mathsf{Va}) : \mathsf{Cache} \times Option\ (\mathsf{Ma} \times \mathsf{Page})$ searches for va in the cpt page table, obtaining the corresponding machine address ma. It then removes the entry (va, ma) from the cache, if it exists. The function returns the new cache and the optional removed entry.

$remove\_cache\_ma(c : \mathsf{Cache}, ma : \mathsf{Ma}) : \mathsf{Cache} \times Option\ (\mathsf{Ma} \times \mathsf{Page})$ if there is an entry (va, ma) in the cache, for any va, this removes it, and returns the corresponding page. Note that since aliased pages are not cached, there can only be one entry in the cache for a given ma. Otherwise, returns the cache unchanged.

$update\_cache\_entry(mem : \mathsf{Memory}, entry : Option\ (\mathsf{Ma} \times \mathsf{Page})) : \mathsf{Cache}$ if $entry$ is some pair (ma, pg), updates the memory with pg as the page in ma (returns $mem[ma := pg]$). Otherwise, returns mem unchanged. This function is used to write back in memory an optional removed cache entry, returned by the $remove\_cache\_va$ and $remove\_cache\_ma$ functions.

$mark\_page\_non\_cacheable : (mem : \mathsf{Memory}, ma : \mathsf{Ma}) : \mathsf{Memory}$ sets the non-cacheable flag to the page in ma.

$tlb\_flush(tlb : \mathsf{TLB}) : \mathsf{TLB}$ returns the empty TLB.

$free\_madd(mem : \mathsf{Memory}) : \mathsf{Ma}$ returns a machine address that is not used by any OS or the hypervisor; so it is suitable for a `page_pin` action.

$madd\_in\_no\_PT(os : os\_ident, ma : \mathsf{Ma}, mem : \mathsf{Memory}) : Prop$ is a predicate that holds when no page table of os maps a virtual address to ma.

$is\_empty : (pg : \mathsf{Page}) : Prop$ is a predicate that indicates whether the given page table is empty.

2.2.2 Valid state

We define a notion of valid state that captures essential properties of the platform. Formally, the predicate $valid\_state$ holds on a state s if s satisfies the following properties:

1. if the active OS is running then no hypercall requested by it is pending;
2. if the hypervisor is *running*, the processor must be in *supervisor* mode, and if an OS is *running*, the processor must be in *user* mode;
3. the hypervisor maps an OS physical address to a machine address owned by that same OS. This mapping is also injective;

4. all page tables of an OS o map accessible virtual addresses to pages owned by o and not accessible ones to pages owned by the hypervisor;

5. the current page table of any OS is owned by that OS;

6. any machine address ma which is associated to a virtual address in a page table has a corresponding physical address as pre-image in the hypervisor mapping;

7. memory pages from different virtual addresses mapped to the same machine address (*synonym* problem) are marked as *non-cacheable*;

8. all cache keys, pairs of the form (va, ma), are related in some page table of the memory;

9. all cache pages have the same owner and type (readable/writable) as those in machine memory;

10. if a virtual address va is translated into a machine address ma according to the TLB, then ma is associated to va in the active memory mapping;

All properties have a straightforward interpretation in our model. The valid states property is invariant under action execution, as shown in Section 2.4.

2.3 Action semantics

The operational semantics of the platform is modelled as a labelled transition system:

$$s \xhookrightarrow{a,o} s'$$

where s, s' range over states, a ranges over actions and o ranges over OS identifiers. Informally, such a transition indicates that the execution of the action a by o from a state s, leads to a new state s'.

The type *Action* is defined as a enumerated type of all modeled platform actions. Figure 2.1 presents the list of actions with their parameters and effects.

Actions can be classified as follows:

1. access, from an OS or the hypervisor, to RW memory pages (`read`, `read_hyper`, `write` and `write_hyper`);

2. update of page tables by the hypervisor on demand of an OS (`new` and `del`);

3. context switches, that change the active OS or the current page table of an OS by the hypervisor (`switch` and `lswitch`);

4. hypervisor call (`hcall`);

5. changes of the execution mode (`chmod` and `ret_ctrl`);

6. changes in the hypervisor memory mapping, which are performed by the hypervisor on demand of an OS (`pin` and `unpin`); and

7. a silent action (`silent`), that models behavior that does not affect the state.

Action	Informal description	Effect
`read` va	Guest OS reads virtual address va	`read` va
`read_hyper` va	The hypervisor reads virtual address va	`read` va
`write` va val	Guest OS writes value val in va	`write` va
`write_hyper` va val	The hypervisor writes value val in va	`write` va
`new` va pa	Hypervisor extends the memory of the active OS with $va \mapsto ma$	`new` va pa
`del` va	Hypervisor deletes mapping for va from the current memory mapping of the active OS	`del` va
`switch` o	Hypervisor sets o to be the active OS	`switch` o
`lswitch` pa	Hypervisor changes the current memory mapping of the active OS to be pa	`lswitch` pa
`hcall` hc	An OS requires privileged service hc to be executed by the hypervisor	`hcall` hc
`ret_ctrl`	Returns the execution control to the hypervisor	`ret_ctrl`
`chmod`	The hypervisor gives the execution control to the active OS	`chmod`
`page_pin` pa t	The memory page that corresponds to pa is registered and classified with type t for the active OS	`page_pin` pa t
`page_unpin` pa	The memory page of the active OS that corresponds to pa is un-registered	`page_unpin` pa
`silent`	Represents the silent action (no effects on the system)	$\emptyset$

Table 2.1: Actions and their effects

Each action a has an effect $\mathsf{eff}(a)$. We use these effects to classify the observations that can be drawn from an action execution. Actions have as effect that same action, except for the write, which only has as the effect parameter the virtual address and not the value; and silent, which has no effect. Note that two actions have the same effect if and only if, the actions are the same action or two writes to the same address but with arbitrary values. In Chapter 3 this notion of effects is used to express the isolation property.

In what follows the semantics of actions will be presented using the following layout:

Action `name` *args*

Informal description of the action behavior

Rule

Formal description of the behavior using natural semantics.

Precondition

Informal description of the conditions that must be satisfied for a correct execution of the action.

Postcondition

Informal description of the effect of the action execution

Notes

Remarks concerning the action

2.3.1 Memory accesses

Memory access actions represent the instructions executed when the operating systems or the hypervisor read or write memory pages. All these actions act on the virtual address space, as is normal in most architectures. The accessed pages are cached (eventually replacing some previous entry), and the TLB is updated with the new entry.

Action `read` va

Guest OS reads the data in va

Rule

$$\frac{\begin{array}{c} act = (aos, running) \qquad os\_accessible(va) \\ get\_page\_mem(s, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \\ cache\_add(cache, va, ma, pg) = (cache', (ma', pg')) \\ mem[ma' := pg'][ma := pg] = mem' \qquad tlb[va := ma] = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{read}\ va,act} (oss, act, hyp, mem', cache', tlb')}$$

$$\frac{\begin{array}{c} act = (aos, running) \qquad os\_accessible(va) \\ get\_page\_mem(s, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \\ cache\_add(cache, va, ma, pg) = (cache', \bot) \qquad tlb[va := ma] = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{read}\ va,act} (oss, act, hyp, mem, cache', tlb')}$$

Precondition

The action `read` va requires that the active OS aos is running, that va is accessible by aos, and that the current page table of aos maps the virtual address va to a machine address ma and a page pg. Moreover, pg is readable/writable.

Postcondition

After the execution of the action, the cache is updated with the entry associated to the pair (va, ma). The TLB is updated with the pair (va, ma).

If an entry is evicted from the cache (first rule), then its page must be the present at the corresponding address in the new memory (either because it already was, in a write-through policy; or because it is written to memory as a result of the eviction, in a write-back policy).

Otherwise (second rule), the memory does not change.

Action `read_hyper` *va*

The hypervisor reads virtual address *va*

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad get\_page\_mem(s, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \\ cache\_add(cache, va, ma, pg) = (cache', (ma', pg')) \\ mem[ma' := pg'][ma := pg] = mem' \qquad tlb[va := ma] = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{read\_hyper}\ va, act} (oss, act, hyp, mem', cache', tlb')}$$

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad get\_page\_mem(s, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \\ cache\_add(cache, va, ma, pg) = (cache', \bot) \qquad tlb[va := ma] = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{read\_hyper}\ va, \bot} (oss, act, hyp, mem, cache', tlb')}$$

Precondition

The action `read_hyper` *va* requires that the active OS *aos* is in waiting mode (or equivalently, that the hypervisor is running). The current page table of the active OS *aos* maps the virtual address *va* to a machine address *ma* and a page *pg*. Moreover, *pg* is readable/writable.

Postcondition

The postcondition of this action is the same as the postcondition of the `read` action.

Action `write` *va val*

Guest OS writes value *val* in *va*

Rule

$$\frac{act = (aos, running) \qquad get\_page\_mem(s, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \qquad cache\_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', (ma', pg')) \qquad mem[ma' := pg'][ma := (RW\ val, OS\ aos, b)]_{pol} = mem' \qquad tlb[va := ma] = tlb'}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write}\ va\ val, act} (oss, act, hyp, mem', cache', tlb')}$$

$$\frac{act = (aos, running) \qquad get\_page\_mem(s, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \qquad cache\_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', \bot) \qquad mem[ma := (RW\ val, OS\ aos, b)]]_{pol} = mem' \qquad tlb[va := ma] = tlb'}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write}\ va\ val, act} (oss, act, hyp, mem', cache', tlb')}$$

Precondition

The action `write` *va val* requires that the active OS *aos* is running. Furthermore, the virtual address *va* is mapped to a machine address *ma* and a readable/writable page *pg* in the current page table of the active OS (*get_page_mem*).

Postcondition

There are two rules for the `write` action, one in which an entry is evicted from the cache when the written page is added, and the other in which no entry is evicted. In both cases the resulting state differs in the value *val* of the page associated to the pair (va, ma) in the cache *cache*, and in the TLB *tlb*. If *cache_add* returns an entry (ma', pg') that was evicted from the cache, the memory in ma' is updated with pg'. The final value in memory of the page in *ma* is dependent on the write policy in use ($mem[ma := page]_{pol}$ updates the page in *ma* with *page* in the write-through policy, and it leaves it unchanged in the write-back case).

Action `write_hyper` *va val*

The hypervisor writes value *val* in *va*

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad get\_page\_mem(s, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \\ cache\_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', (ma', pg')) \\ mem[ma' := pg'][ma := (RW\ val, OS\ aos, b)]_{pol} = mem' \qquad tlb[va := ma] = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write\_hyper}\ va\ val, \bot} (oss, act, hyp, mem', cache', tlb')}$$

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad get\_page\_mem(s, va) = (ma, pg) \qquad pg = (RW\ \_, OS\ aos, b) \\ cache\_add(cache, va, ma, (RW\ val, OS\ aos, b)) = (cache', \bot) \\ mem[ma := (RW\ val, OS\ aos, b)]]_{pol} = mem' \qquad tlb[va := ma] = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{write\_hyper}\ va\ val, \bot} (oss, act, hyp, mem', cache', tlb')}$$

Precondition

The action `write_hyper` *va val* requires that the hypervisor is running. Furthermore, the virtual address *va* is mapped to a machine address *ma* and a readable/writable page *pg* in the current page table of the active OS (*get_page_mem*).

Postcondition

The postcondition of this action is the same as the postcondition of the `write` action.

2.3.2 Page table updates

Page table updates modify the operating system page tables adding or removing mapping entries. Only the hypervisor can execute these actions on behalf of an operating system. If a guest operating system wants to modify one of its page tables, it has to make a hypercall with the corresponding parameters (see section 2.3.4).

Action `new` *va pa*

Hypervisor extends the virtual memory of the active OS with $va \mapsto ma$

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \quad os\_accessible(va) \quad oss[aos] = (pa', New\ va\ pa) \\ get\_page\_hyp(s, aos, pa) = (ma, pg) \quad \neg memory\_alias(mem, va, ma) \\ get\_page\_hyp(s, aos, pa') = (ma', cpt) \quad cpt[va := ma] = cpt' \\ oss[aos := (pa', None)] = oss' \quad tlb[va := \bot] = tlb' \\ remove\_cache\_va(cache, cpt, va) = (cache', entry) \\ update\_cache\_entry(mem, entry) = mem' \quad mem'[ma' := cpt'] = mem'' \end{array}}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{new}\ va\ pa, act} (oss', act, hyp, mem'', cache', tlb')}$$

$$\frac{\begin{array}{c} act = (aos, waiting) \quad os\_accessible(va) \quad oss[aos] = (pa', New\ va\ pa) \\ get\_page\_hyp(s, aos, pa) = (ma, pg) \quad memory\_alias(mem, va, ma) \\ get\_page\_hyp(s, aos, pa') = (ma', cpt) \quad cpt[va := ma] = cpt' \\ oss[aos := (pa', None)] = oss' \quad tlb[va := \bot] = tlb' \\ remove\_cache\_va(cache, cpt, va) = (cache', entry) \\ update\_cache\_entry(mem, entry) = mem' \\ remove\_cache\_ma(cache', ma) = (cache'', entry') \\ update\_cache\_entry(mem', entry') = mem'' \\ mem''[ma' := cpt'] = mem''' \quad mark\_page\_non\_cacheable(mem''', ma) = mem'''' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{new}\ va\ pa, act} (oss', act, hyp, mem''', cache'', tlb')}$$

Precondition

The action `new` *va pa* requires that the hypervisor is running, that *va* is accessible by the active OS, and that *aos* has a pending new hypercall. Furthermore, *pa* should be mapped to some machine address *ma* and page *pg*.

Postcondition

In the state after the execution, the current page table *cpt* is updated with the mapping of *va* to *ma*. *va* could appear mapped to some other machine address ma' in the current page table, if the operating system is requesting an update of the existing mapping $va \mapsto ma'$ to $va \mapsto ma$. In this case, the entry corresponding to (va, ma') is removed from the cache by the *remove_cache_va* function, and the old cached page value is written to memory.

If the new address is an alias (second rule), in addition to the above, the cache entry with *ma* as its address is removed from the cache (and written to memory). Additionally, the page in *ma* is marked as non cacheable.

In both cases, the mapping from *va* is removed from the TLB, if it exists.

Action `del` *va*

Hypervisor deletes mapping for *va* from the current page table of the active OS

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad os\_accessible(va) \qquad oss[aos] = (pa', Del\ va) \\ get\_page\_hyp(s, aos, pa') = (ma', cpt) \\ oss[aos := (pa', None)] = oss' \qquad cpt[va := \bot] = cpt' \qquad mem[ma' := cpt'] = mem' \\ remove\_cache\_va(cache, cpt, va) = (cache', \_) \qquad tlb[va := \bot] = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{del}\ va,act} (oss', act, hyp, mem', cache', tlb')}$$

Precondition

The action `del` *va* requires that the hypervisor is running, that *va* is accessible by the active OS *aos*, and that *aos* has a pending del hypercall.

Postcondition

In the resulting state, the current page table *cpt* of *aos* is updated by deleting the mapping from *va*. Additionally, if *va* maps to some *ma* in the current page table of *aos* and there exists an entry in the cache for (va, ma), this entry is removed. Similarly, the mapping from *va* is removed from the TLB.

In this case, the eventual entry removed from the cache need not be written back to memory, since after the deletion in the *cpt*, the corresponding page is now inaccessible. Indeed, a cacheable page is never aliased, and therefore there is only one page table that maps some virtual address to this page, and that is the one this action is deleting.

2.3.3 Context switches

Context switches occur when the currently running process in the processor is switched to let another execute. We define two kinds of context switches: operating system or global context switch (`switch`, sometimes called *world switch*) in which the hypervisor selects another operating system to execute, and local or application context switch (`lswitch`) in which the running application inside the active operating system is changed.

Action `switch` o

Hypervisor sets o to be the active OS

Rule

$$\frac{act = (aos, waiting) \qquad oss[o] = (pa, None) \\ (o, waiting) = act' \qquad tlb\_flush(tlb) = tlb'}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{switch}\ o,o} (oss, act', hyp, mem, cache, tlb')}$$

Precondition

The context `switch` o action requires that the hypervisor is running, and that there is no hypercall pending for the OS o.

Postcondition

In the resulting state, o is the new active OS, and the TLB tlb is (fully) flushed.

Action `lswitch` pa

Hypervisor changes the current memory mapping of the active OS to be pa

Rule

$$\frac{act = (aos, waiting) \qquad oss[aos] = (\_, LSwitch\ pa) \\ get\_page\_hyp(s, aos, pa) = (ma, pg) \qquad pg = (PT\ \_, \_, \_) \\ oss[aos := (pa, None)] = oss' \qquad tlb\_flush(tlb) = tlb'}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{lswitch}\ pa,act} (oss', act, hyp, mem, cache, tlb')}$$

Precondition

The action `lswitch` pa requires that the hypervisor is running, and that the active OS is waiting for an hypercall to change its current memory mapping to pa. Additionally, there exists a machine address ma, associated by the hypervisor mapping to the physical address pa, which is the address of a PT page.

Postcondition

The effect of this action is to set pa as the address of the current page table of the active OS; and flush the TLB.

2.3.4 Hypercall

Hypercalls are an API exposed by the hypervisor to the operating systems, in order for them to be able to perform privileged operations on the platform. The hypervisor, when receiving an hypercall, first validates its parameters, and then performs the required action on behalf of the operating system.

Action `hcall` *hc*

The active OS requires privileged service *hc* to be executed by the hypervisor.

Rule

$$\frac{act = (aos, running) \qquad oss[aos] = (pa, \_) \qquad oss[aos := (pa, hc)] = oss' \qquad (aos, waiting) = act'}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{hcall}\ hc,act} (oss', act', hyp, mem, cache, tlb)}$$

Precondition

The action `hcall` *hc* requires that the active OS is running.

Postcondition

In the resulting state, the active OS requires the execution of *hc* to the hypervisor, which takes control of the execution.

2.3.5 Changes of execution mode

The `ret_ctrl` and `chmod` actions modify the execution mode of the processor. The first instruction happens when the hypervisor regains control of the processor from an operating system (either because the operating system gives back control voluntarily or because of an interruption). The second one happens when the hypervisor is ready to let some operating system execute.

Action `ret_ctrl`

The active OS returns the execution control to the hypervisor.

Rule

$$\frac{act = (aos, runnig) \qquad (aos, waiting) = act'}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{ret\_ctrl},act} (oss, act', hyp, mem, cache, tlb)}$$

Precondition

The action `ret_ctrl` requires that the active OS is running.

Postcondition

In the resulting state, the hypervisor is running.

Action `chmod`

The hypervisor gives to the active OS the execution control.

Rule

$$\frac{act = (aos, waiting) \qquad oss[aos] = (\_, None) \qquad (aos, running) = act'}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{chmod},act} (oss, act', hyp, mem, cache, tlb)}$$

Precondition

The action `chmod` requires that the hypervisor is running, and that there is no hypercall pending for the active OS.

Postcondition

In the resulting state, the active OS is running.

2.3.6 Hypervisor mapping updates

Hypervisor mapping updates model dynamic memory allocation and deallocation by operating systems. They are privileged actions, and are therefore only executed by the hypervisor (on behalf of an operating system, by means of an hypercall). These updates allocate a free memory page to the requesting operating system (`page_pin`), or deallocates a page from its memory (`page_unpin`).

Action `page_pin` *pa t*

The memory page that corresponds to physical address *pa*, for the active OS, is registered and classified with type *t*.

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad oss[aos] = (pa', Pin\ pa\ t) \\ hyp[aos][pa] = \bot \qquad free\_madd(mem) = ma \\ oss[aos := (pa', None)] = oss' \qquad hyp[aos][pa := ma] = hyp' \\ mem[ma := (t, aos, true)] = mem' \end{array}}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{page\_pin}\ pa\ t,act} (oss', act, hyp', mem', cache, tlb)}$$

Precondition

The action `page_pin` *pa t* requires that the hypervisor is running, that the active OS must be waiting for an hypercall to *pin* the physical address *pa* of type *t*, and that *pa* must not be already allocated. In addition to that, there must be a free machine memory (*ma*) available.

Postcondition

The effect of the action is to create and allocate at machine address *ma* a new page of type *t* whose owner is the active OS and bind, in the hypervisor mapping, the physical address *pa* to *ma*. The rest of the state remains unchanged.

Action `page_unpin` *pa*

The memory page that corresponds to physical address *pa*, for the active OS, is un-registered.

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad oss[aos] = (pa', UnPin\ pa) \qquad pa' \neq pa \\ get\_page\_hyp(s, aos, pa) = (ma, pg) \\ is\_empty(pg) \qquad madd\_in\_no\_PT(aos, ma, mem) \\ oss[aos := (pa', None)] = oss' \qquad hyp[aos][pa := \bot] = hyp' \\ mem[ma := (Other, No\_Owner, true)] = mem' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{unpin}\ pa,act} (oss', act, hyp', mem', cache, tlb)}$$

Precondition

The action `page_unpin` *pa* requires that the hypervisor is running, that the active OS must be waiting for an hypercall to *unpin* the physical address *pa*, that *pa* is not the current page physical address of the active OS, and that if the page associated with the machine address *ma*, mapped by *pa* in the hypervisor mapping, is of type *PT*, it must be empty. Finally, *ma* can not appear in any page table of the active OS (all such mappings should be removed beforehand, by calling the `del` action).

Postcondition

The action removes the mapping of *pa* to *ma* in the hypervisor mapping. In the resulting state, the page *pg* is un-registered.

2.3.7 Silent

The silent action is any instruction that does not have an effect on the state of the platform. One instance of such an instruction might be an arithmetic operation on registers, which does not modify the memory state of the platform.

In addition, we have used silent actions for some properties such as transparency in [21], where we proved that operating system behavior does not change if we turn all actions of other operating systems into the `silent` action.

Action `silent`

Represents the silent action –the system does not advertise any effects.

Rule

$$\frac{}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{silent}, act} (oss, act, hyp, mem, cache, tlb)}$$

Precondition

This operation has no restrictions for execution.

Postcondition

The state does not change.

2.4 Invariance of valid state

Given the state definition and action semantics, we can now prove that one-step execution preserves the valid state property. That is to say, the state resulting from the execution of an action in a valid state is also a valid one.

Lemma 2.4.1 (Valid State Invariant)**.**
$\forall\ (s\ s' : \mathsf{SLST})\ (a : Action)\ (o : \mathsf{OSId}),\ valid\_state(s) \rightarrow\ s \xrightarrow{a,o} s'\ \rightarrow\ valid\_state(s')$

Proof. To prove this lemma one has to prove all valid state conditions for all actions. We will briefly describe the proof for some of these cases.

- First, we start with conditions 8 and 9 for the `write` action. Condition 8 states that "all cache keys, (va, ma) pairs, are related in some page table mapping of the memory"; and condition 9 that "all cache pages have the same owner and type (readable/writable) as those in machine memory".

  We assume we have a valid state s and execute the `write` va_w val, to get to state s'. By the semantics of the `write` action, there must be a machine address ma_w such that va_w maps to ma_w in the current page table of the active OS. If the written page is not cacheable, the cache does not change in s'. The only change is in the value of the memory page at ma_w.

  Since this page is not cacheable, no cache entry can have ma_w as a machine index, and therefore condition 8 and 9 holds for all cache pages in s'.

  If the written page is cacheable, then in s' there is a new entry (va_w, ma_w) in the cache. As mentioned before, the current page table mapping maps va_w to ma_w in s and s', proving condition 8. Furthermore, this page have the same owner and type as the memory page, since the only difference is in the value; therefore condition 9 also holds.

- The other case we will consider is that of conditions 4 and 6 for the `new` action. These conditions state that "all page tables of an OS o map accessible virtual addresses to pages owned by o and not accessible ones to pages owned by the hypervisor" and "any machine address ma which is associated to a virtual address in a page table has a corresponding physical address as pre-image in the hypervisor mapping", respectively.

  Again, we assume we have a valid state s and execute the `new` va_n pa_n, to get to state s'; where we know that va_n is accessible. If the operating system o is not active, the

conditions hold since it is another operating system that executes the action, and therefore the memory structures of o do not change. If it is active, then by the action semantics, pa_n must have a corresponding ma_n in the hypervisor mapping of o; and therefore condition 6 holds. Furthermore, since s is valid and the hypervisor mapping does not change, the memory page at ma_n must be owned by o, proving condition 4.

In this manner we can prove all conditions for all actions to finish the proof of the lemma. It should be noted that in the previous description many details have been omitted. In reality, the proof of this lemma is particularly cumbersome, involving the analysis of many cases, derived from the complexity of the action semantics and the valid state conditions. □

Platform state invariants, such as state validity, are useful to analyze other relevant properties of the model. In particular, the results presented in the following sections are obtained from valid states of the platform.

Chapter 3

Isolation

One of the main goals of this work is to provide convincing and formal arguments that constant-time implementations constitute an adequate defense against the threats posed by cache-based attacks. For this we use the idealized model of virtualization presented in the previous chapter, where for the sake of concreteness, we will consider a scenario with only two guest operating systems: a victim operating system o_v and an attacker operating system o_a. In particular, we will focus on the class of attacks where the malicious operating system o_a adaptively probes the state to retrieve information about previous execution steps and then launches an attack to learn information otherwise private to the victim operating system. The description of this attacker model will be presented in Section 3.1.

The property of isolation and non-leakage of victim data in which we are interested is a non-interference result on execution traces of the platform, where the notion of indistinguishability amounts to equality of observations resulting from the execution of the victim actions. In Section 3.2 we describe the general framework for reasoning about non-interference and its variants. Following the usual non-interference definitions, in Section 3.3 we define the state equivalence relation we will use. In Section 3.4 we introduce execution traces; then we show the necessary unwinding lemmas in Section 3.5, and prove our main isolation result in Section 3.6.

3.1 Adversary model

The class of attacks we model are cache timing attacks, i.e. attacks that exploit the side-channel created by the timing difference between a data access from the cache (cache hit) and an access from main memory (cache miss). This kind of attacks have been successfully used to recover the secret keys of various cryptographic algorithms, such as DES [121] and AES [106, 68].

For efficiency, usual implementations of cryptographic algorithms make use of precomputed results stored in memory called look-up tables. Instead of computing a time-consuming operation in real-time, the result is searched in those tables. The problem arises because the parameters of the operation (the indices used to access the table) generally depend on the secret key used. If an attacker could see the sequence of memory accesses of the victim program, he could deduce the value of the key, since he knows which table entries were accessed.

Even before these attacks, the problem of side-channels was a well-known issue that led to their integration into security models for cryptography. Physically observable cryptography [99] is one example of a model that aims at providing a realistic framework for cryptographic proofs by giving adversaries the ability to draw observations about the physical execution of programs. One essential component of physically observable cryptography is a leakage oracle that can be called by the adversary with an observation function ℓ from states to some observation domain Obs, and obtains as a response the value of $\ell(s)$ where s is the current state of the system. The

leakage function captures at an abstract level and in a uniform manner different side-channels and yields a tractable model in which to reason about the security of cryptographic constructions, in the style of provable security.

We specialize physically observable cryptography to the case of cache-timings attacks, in which the attacker can perform timing measurements on the cache in order to discover which of his cached pages are no longer cached after execution of the victim OS. The observations are therefore the set of virtual addresses that correspond to evicted cache entries of the attacker. Given a transition $s \xrightarrow{a,o} s'$, we will define the function $\ell(s, s')$ as the set of virtual addresses of the attacker that are cached in s, but evicted during the execution of action a by the OS o, and therefore not cached in s'.

The security property we prove establishes that the observations that the adaptive adversary draws during execution of a constant-time victim are independent of the victim data. The adaptive attacker in this setting is formalized using an adversarially controlled scheduler: at each step the adversary can decide whether the victim operating system or itself will execute. We further give the adversary the power to trigger the resolution of pending hypercalls. Formally, we model this attacker as a function $\mathcal{A}$ that takes a partial execution trace up to that point, and returns either a tag $Victim$, if the attacker lets the victim operating system perform the next action, or an action of its choice *Attacker a* that it will execute in the next step. The reason for giving the attacker complete control over the scheduler is to model the ability of the attacker to let the victim execute few instructions between measurements, improving the bandwidth of the side-channel. For example, Bangerter, Gullash and Krenn show an efficient attack on AES, by exploiting the Linux *Completely Fair Scheduler* [68].

In addition to controlling the scheduler and observing its cache entries, we will let the attacker observe all state components, except the values stored in readable/writable pages owned by the victim. We will therefore define the attacker so that it will not be able to distinguish between states that only differ in victim data. In other words, the adversary will be able to observe its own readable/writable memory pages; the values of its own cache entries; the memory layout of the victim, as defined by the page metadata (owner and cacheable status) of the victim memory pages and the layout and history of the cache. The adversary cannot, however, directly read, write or observe page tables or the hypervisor mappings (either its own or the victim). This is because these mappings are maintained by the hypervisor and guest OSs have no direct access to them. Moreover, the adversary cannot directly observe the values held in the memory or cache entries of the victim.

By giving the adversary this much visibility over the victim state, we capture the kind of attacks we are interested in: if two states differ in one of these observable components, the execution of an action might replace an attacker entry in the cache, potentially leading to a cache-based information leakage. On the other hand, we will prove that if two actions with the same effect are executed in two equivalent states (indistinguishable for the attacker), the attacker cache entries are equal in the resulting states, and therefore the leakage function will always return the same observations.

Our adversary model is strong enough to allow us to model various kinds of realistic attacks. For example, Osvik et al [106] use synchronous attacks to efficiently recover AES keys. In a synchronous attack the attacker can trigger victim encryptions at will. Our adversary is able to perform these attacks, since it has complete control over the scheduler. The authors mention two different kinds of synchronous attacks: "Evict+Time" and "Prime+Probe".

In the first, "Evict+Time", the attacker begins by executing the encryption once, making sure the victim relevant look-up table entries are stored in the cache. Then it evicts some cache entries by accessing its own memory addresses, and runs the encryption once more, this time timing its execution. By measuring the victim execution time, the attacker can learn whether the

evicted entry was accessed by the second execution or not. This allows the attacker to recover the key by repeating this process and performing a statistical analysis afterwards.

We are able to capture this kind of attacks. Indeed, if we consider two executions of the attack, such that the cache entry evicted by the attacker is accessed in the first but not in the second, then our adversary would be able to draw different observations. This is because after the first execution, the victim replaces an attacker entry in the cache (therefore the leakage function returns this entry virtual address); while after the second execution, no attacker entry is replaced (and therefore the leakage function returns no virtual address).

The second kind of attack, "Prime+Probe", can also be modeled. This attack works by first filling the cache with attacker entries (by reading a wide range of memory address); then triggering an encryption, where the victim replaces some (attacker) entries from the cache; and finally measuring the time it takes for the attacker to access its own addresses. In this case, our leakage function would return different results during the execution of the victim, if it removes different attacker entries from cache.

3.2 Non-interference

To prove that the behavior of the attacker is not influenced in any way by private data of the victim, we need to study how information flows between guest operating systems in our model, even in the presence of cache side-channels. For this we will use a technique called *non-interference*, first described by Goguen and Meseguer [63, 64], which is a formal framework for the specification and verification of security properties based on restrictions on the information flow between different *domains* of a system. A domain is a portion of the system with a given security level. It can be a user, a process or as in our setting, a guest operating system.

The basic idea is to separate the model of a system from the security policies specifications that the system should exhibit, and to provide a general technique for proving that a system satisfies a given policy. The model is defined in terms of state transitions, with focus on operations and outputs. The policies must specify the restrictions on how the different domains interact with one another, and are therefore defined in terms of: interference assertions, which expresses that a domain u can influence another domain v (written $u \rightsquigarrow v$); and non-interference assertions, that prohibit a domain from influencing another (written $u \not\rightsquigarrow v$) [111].

We will closely follow von Oheimb's formulation of non-interference [103], using similar concepts, and instantiating various security models defined in that work to study variations that are relevant for our purposes. In that work, two variants of the model are presented for deterministic and nondeterministic systems. Since our virtualization model is deterministic (see Section 2.1.2), we will focus only on the deterministic case. In this case, a *step* function is defined so that given a state and an action, it returns the new state after the execution of the action. A *run* of a system is defined as the lifting of the *step* function to a finite sequence of actions from an initial state s_0. Therefore, $run(s_0, \alpha)$ is the final state after executing all actions in the list α. Furthermore, each action a has a domain associated with it (written $dom(a)$), that represents the domain that executes the action.

There are basically two complementary policies that show that a domain does not influence another, depending on whether the focus is on behavior (some events are secret and should not interfere with other domains) or on data (some data is secret and should not leak to untrusted domains). In the first case, the attacker must be unable to distinguish whether a secret event happened or not; while in the second, the attacker is unable to distinguish between two executions starting from states that only differ on the secrets. This lead von Oheimb to define two related notions for each of these approaches: *non-interference* and *non-leakage*, and then combine them in what he calls *non-influence*.

Non-interference captures the idea that a domain v cannot distinguish between an execution with a domain u that cannot influence it ($u \not\rightsquigarrow v$) and an execution where the actions of u are not present. In order to model the ability of a domain to distinguish between executions, an *unwinding* relation on states parameterized by a domain is defined. This relation captures the partial view of the system by this domain. Two states s and t are in this relation, written $s \overset{u}{\sim} t$, if u cannot distinguish between the two. In the general case, this does not need to be an equivalence relation [96]. In the following, we will only consider state equivalences as the unwinding relation, so we will also restrict this discussion to these kinds of relations.

Formally, the non-interference property can be stated as follows:

Definition 3.2.1 (Non-interference)**.** A system is non-interferent if and only if for every initial state s_0, list of actions α, and domain u, $run(s_0, \alpha) \overset{u}{\sim} run(s_0, ipurge(u, \alpha))$.

Here, the function *ipurge* removes all actions executed by a domain that cannot influence u, either directly or indirectly, from the run. This property shows that domains cannot distinguish between a run and the variant without the actions of the domains not allowed to interfere with it. This implies that the execution of these actions have no effect on the behavior of u, and therefore, no interference is possible between these domains.

Two lemmas are used to prove this property; these are the *step consistent* and the *locally preserves* unwinding lemmas:

Lemma 3.2.2 (Step-consistent unwinding lemma)**.** Let s and t be states, a an action and u a domain, such that $s \overset{u}{\sim} t$. Then, $step(a, s) \overset{u}{\sim} step(a, t)$.

Lemma 3.2.3 (Locally preserves unwinding lemma)**.** Let s be a state, u a domain and a an action, such that $dom(a) \not\rightsquigarrow u$. Then, $s \overset{u}{\sim} step(a, s)$.

The first lemma establishes that the execution of the actions allowed to interfere with u depends exclusively on the visible part of the state. The second, that the execution of the other actions are transparent for u, in the sense that u cannot tell if the action was executed or not. With these two lemmas, the non-interference property is established.

There is also a "strong" version that allows for arbitrary action removal or addition, instead of purging all actions not allowed to interfere with a domain. The property in this case would state that, given two arbitrary runs α and β, such that $ipurge(u, \alpha) = ipurge(u, \beta)$, then $run(s_0, \alpha) \overset{u}{\sim} run(s_0, \beta)$. However, it is easy to show that this version is equivalent to the classical one when the unwinding relation $\sim$ is symmetric and transitive.

Note that both versions use finite runs of actions, and they only require that the equivalence holds in the final states. In other words, the system runs to the end and then the resulting states are inspected to see if there are observable differences. Our approach will deviate from these assumptions, and use infinite (co-inductive) runs. The reason for this is that we are modeling the execution of a virtualization platform and there is no final state to be reached, the platform should continuously execute, handling requests from the processes in the guest operation systems.

This has important implications on the way we can state noninterference properties. Since it is somewhat complex to define a purging function on infinite runs, we will define an equivalence relation that constructs two traces with the same visible actions, and arbitrary non-visible actions. This is a variation on "strong" non-interference, that uses a co-inductive equivalence relation for the condition $ipurge(u, \alpha) = ipurge(u, \beta)$. We present the formal definitions of these relations in Section 3.4 and Chapter 5. In a previous work we have used and proved a non-interference property more similar to the classical one, which we called *transparency*, where instead of purging, we transform non-observable actions to the `silent` action [21].

The classical notion of non-interference is primarily focused on the visibility of events. The secrecy of state information is secondary, and handled via side-effects. There is a complementary version, called *non-leakage*, that deals with secret information and treats events as secondary.

This property can be stated as follows:

Definition 3.2.4 (Non-leakage)**.** Let s and t be a states, u a domain and α a list of actions, such that $s \overset{sources(u,\alpha)}{\sim} t$. The system is non-leaking if and only if $run(s,\alpha) \overset{u}{\sim} run(t,\alpha)$.

Here the function $sources(u,\alpha)$ returns the domains that execute actions in α that can influence u.

Non-leakage means that, starting from two indistinguishable states, the execution of a sequence of actions leads to indistinguishable final states; in other words, all possible observations are independent of state information that cannot be observed.

To prove this theorem, only the step-consistent unwinding lemma is necessary, since the same action sequence is used in both executions, and no purging is necessary.

The combination of non-interference and non-leakage, called *non-influence* by von Oheimb, makes it possible to reason about state information and events at the same time. It can be stated as:

Definition 3.2.5 (Non-influence)**.** Let s and t be a states, u a domain and α a list of actions, such that $s \overset{sources(u,\alpha)}{\sim} t$. A system is non-influencing if and only if $run(s,\alpha) \overset{u}{\sim} run(t, ipurge(u,\alpha))$.

In this chapter we will prove a non-leakage result, where the equivalence of states will treat the victim operating system data as secret and the rest of the state as visible. This will allow us to show that constant-time implementations (in which the sequence of actions is fixed and does not depend on data) do not leak information through cache side-channels.

Later, when we introduce stealth memory in Chapter 5, we will not only focus on non-leakage of secret information, but also on the secrecy of stealth actions. We will therefore use a non-influence property in that case.

3.3 Equivalence of states

In order to instantiate non-leakage in our model, we start by defining an equivalence relation $\sim$ between states to represent the partial view of the attacker operating system o_a on the state; thus, two states s and s' s.t. $s \sim s'$ coincide on all parts of the state exposed to the attacker. The fact that we allow dynamic memory allocation through the execution of the `pin` action complicates the definition of this relation. Since we cannot state it directly in terms of machine addresses, we do it indirectly through the use of the hypervisor mapping physical addresses, which are chosen by the operating systems, and are therefore the same in both executions. The definition of this relation is a conjunction of equivalence definitions for the relevant components of the state, which follow:

Definition 3.3.1 (Equivalence of OS information)**.** The operating system information oss and oss' are equivalent for the attacker, if the attacker has the same current page table, and the same hypercall in both states. Furthermore, the attacker must be either active in both states and have the same activity, or not active in the states.

For the equivalence of the memory we consider two cases: when a physical address is mapped to a readable/writable page (Equivalence of hypervisor mappings), and when it is mapped to a page table (Equivalence of cache and memory mappings).

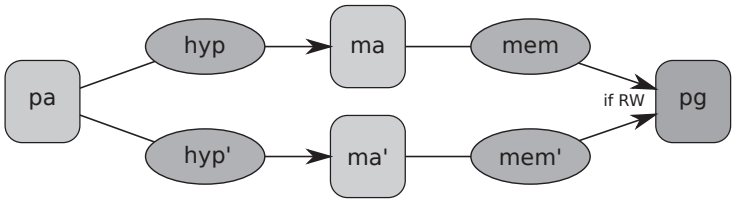

Figure 3.1: Equivalence of hypervisor mappings

In the first case (depicted in Figure 3.1) we require that the attacker readable/writable pages are the same. Furthermore, the metadata of the memory pages of the victim must be the same (pages should have the same owner, and same cacheable flag, but can hold an arbitrary value).

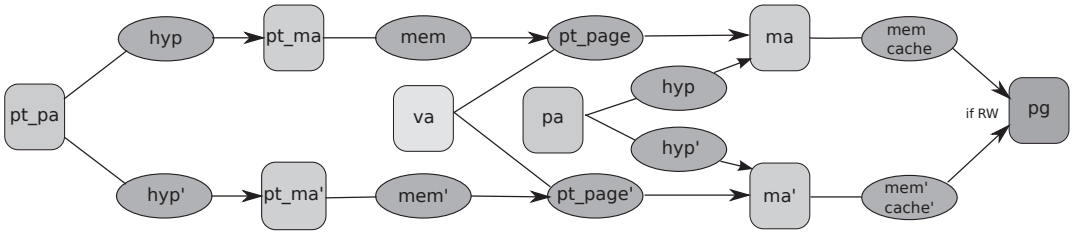

Figure 3.2: Equivalence of cache and memory mappings

In the second, since machine addresses are arbitrary in both states, the page tables will contain different mappings. For these page tables to be in the equivalence relation, all virtual addresses mapped in one page table must be also mapped in the other. Furthermore, readable/writable pages mapped by a virtual address should be the same, if the owner is the attacker, or have the same metadata if it is a victim page (see Figure 3.2).

More formally, these two equivalence relations are as follows:

Definition 3.3.2 (Equivalence of hypervisor mappings)**.** Two states s and s' have equivalent hypervisor mappings for the attacker ($s \sim^{hyp} s'$) if for every physical address pa, readable/writable page pg and machine address ma:

- if $get\_page\_hyp(s, o_a, pa) = (ma, pg)$, there exists ma' such that $get\_page\_hyp(s', o_a, pa) = (ma', pg)$;
- if $get\_page\_hyp(s, o_v, pa) = (ma, pg)$, then there exists ma' such that $get\_page\_hyp(s', o_v, pa) = (ma', pg')$, where pg and pg' are equal except in their contents;

and reciprocally for s'.

Definition 3.3.3 (Equivalence of cache and memory mappings)**.** Two states s and s' have equivalent cache and memory mappings for the attacker ($s \sim^{mem} s'$) if for every physical address pt_pa, page table pt_pg and machine address pt_ma, such that $get\_page\_hyp(s, o, pt\_pa) = (pt\_ma, pt\_pg)$, there exist pt_ma' and pt_pg' such that $get\_page\_hyp(s', o, pt\_pa) = (pt\_ma', pt\_pg')$ and for all virtual addresses va and machine address ma such that $pt\_pg[va] = ma$ and $get\_page\_va\_ma(s, va, ma)$ is readable/writable, then there exists ma' such that $pt\_pg'[va] = ma'$. Furthermore, the same physical address pa that maps to ma in s, maps to ma' in s'; and moreover:

- if $o = o_a$ then $get\_page\_va\_ma(s, va, ma) = get\_page\_va\_ma(s, va, ma')$;

- if $o = o_v$, then $get\_page\_va\_ma(s, ma)$ and $get\_page\_va\_ma(s', ma')$ are equal except in their contents;

and reciprocally for s'.

Finally, we consider the equivalence of cache histories. As stated in Section 2.2 the cache history represent historic access information used by the replacement policy. Two histories are equivalent if they have the same information for all cache indices. This equivalence guarantees that the replacement policy will select the same entry for eviction from the two caches when adding an entry in the same cache index. If this was not the case, it could remove an attacker entry in one state, and a victim one in the other, leading to a leakage of information.

Definition 3.3.4 (Equivalence of cache histories)**.** Two cache histories $history$ and $history'$ are equivalent for the attacker if for all cache index i, $history[i] = history'[i]$.

3.4 Execution traces

The isolation property is eventually expressed on execution traces, rather than execution steps. An execution trace is defined as a stream (an infinite list) of states that are related by the transition relation $\hookrightarrow$, i.e. an object of the form $s_0 \xhookrightarrow{a_0,o_0} s_1 \xhookrightarrow{a_1,o_1} s_2 \xhookrightarrow{a_2,o_2} s_3 \ldots$ such that every execution step $s_i \xhookrightarrow{a_i,o_i} s_{i+1}$ is valid. Formally, an execution trace is defined as a stream Θ of pairs of states, actions and OS, such that for every $i \geq 0$, $s_i \xhookrightarrow{a_i,o_i} s_{i+1}$, where $\Theta[i].st = s_i$, $\Theta[i].act = a[i]$ and $\Theta[i].osi = o_i$.

We lift the indistinguishability relation to execution traces Θ and Θ' using the following co-inductive rules:

$$\frac{s \sim s' \quad \Theta \sim \Theta'}{(s \xhookrightarrow{a,o_a} \Theta) \sim (s' \xhookrightarrow{a,o_a} \Theta')}$$

$$\frac{s \sim s' \quad \Theta \sim \Theta' \quad \mathsf{eff}(a) = \mathsf{eff}(a')}{(s \xhookrightarrow{a,o_v} \Theta) \sim (s' \xhookrightarrow{a',o_v} \Theta')}$$

These rules state that given two equivalent traces, one can add equivalent states s and s' to the beginning, as long as the same attacker action, or victim actions with the same effect are used. In other words, two equivalent traces are indistinguishable for the attacker, at every point of execution.

As mentioned in Section 3.1, the attacker has control not only on the actions it executes (as we are modeling an adaptive attacker) but also on the scheduler, in the sense that it is the attacker who decides what operating system will execute next at any given point of the execution.

We model the attacker controlled scheduler as a function that given the partial trace executed up to that point, returns the next attacker action to execute, or hands control to the victim.

$$\mathcal{A} : partial\_trace \to \{Victim \mid Attacker\ a\}$$

We only require that the result of this function is the same when two partial traces are indistinguishable for the attacker, i.e. when they are equivalent with respect to the $\sim$ relation.

The execution traces will be generated by merging the attacker and victim actions in the way indicated by this scheduler. We introduce the interleaving relation for this purpose:

$$\frac{\mathcal{A}(pt) = Victim \quad s \xhookrightarrow{a,o_v} \Theta[0].st \quad interleave(\Theta[0].st, pt \xhookrightarrow{a,o_v} s, v\_str, \Theta)}{interleave(s, pt, a :: v\_str, s \xhookrightarrow{a,o_v} \Theta)}$$

$$\frac{\mathcal{A}(pt) = Attacker\ a \quad s \xhookrightarrow{a,o_a} \Theta[0].st \quad interleave(\Theta[0].st, pt \xhookrightarrow{a,o_a} s, v\_str, \Theta)}{interleave(s, pt, v\_str, s \xhookrightarrow{a,o_a} \Theta)}$$

Given a state s, a partial trace pt, a stream of victim action v_str, and an execution trace Θ, the interleaving is the result of adding the execution of the next action (of either the victim in the first rule, or the attacker in the second) to the interleaving of the rest of the trace, where the state is appended to the partial trace.

This definition is co-inductive, but it is complicated by the use of the partial traces, which are inductive lists. Intuitively, the result is a trace where attacker actions (chosen by himself based on the partial trace) are merged into the execution of the victim stream of actions.

This abstract interleaving function can be instantiated for common particular cases where the attacker does not have control of the scheduler. As an example, we developed an implementation of a round-robin scheduler and showed that it behaves as modeled by the definitions above.

3.5 Unwinding lemmas

The equivalence relation $\sim$ is kept invariant when the same attacker action or two victim actions with the same effect are executed in two equivalent states. These results are variations of standard non-interference unwinding lemmas, specifically step-consistent unwinding lemmas. Figure 3.3 shows a graphical representation of these lemmas.

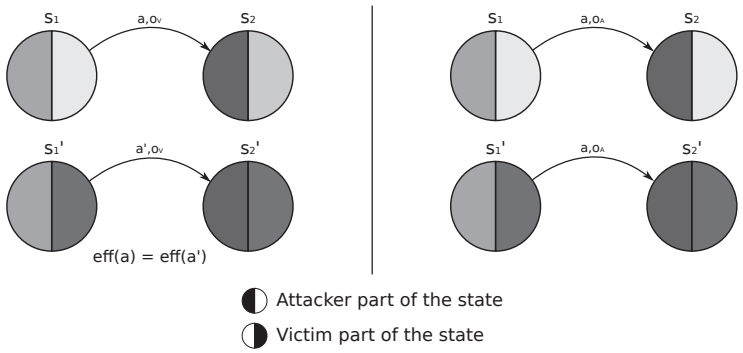

Figure 3.3: Step-consistent unwinding lemmas

Basically, we prove that the execution from two equivalent states of two victim actions with the same effect does not change the attacker part of the state, and that it changes the victim part of the state in a way that preserves the equivalence. We show a similar result when the attacker executes the same action from two equivalent states. Formally:

Lemma 3.5.1 (o_a step-consistent unwinding lemma)**.** Let s_1 and s_1' be states where o_a is active, and such that $s_1 \xrightarrow{a,o_a} s_2$, $s_1' \xrightarrow{a,o_a} s_2'$ and $s_1 \sim s_1'$. Then, $s_2 \sim s_2'$.

Lemma 3.5.2 (o_v step-consistent unwinding lemma)**.** Let s_1 and s_1' be states where o_v is active, and such that $s_1 \xrightarrow{a,o_v} s_2$, $s_1' \xrightarrow{a',o_v} s_2'$, $\mathsf{eff}(a) = \mathsf{eff}(a')$ and $s_1 \sim s_1'$. Then, $s_2 \sim s_2'$.

Proof. To prove these lemmas each action must be analyzed checking the equivalence conditions on the resulting states. To give an idea of the proof, we will describe the case of a write action by the victim for the equivalence of hypervisor mappings.

We start with two valid and equivalent states s_1 and s_1'; two virtual addresses va and va' and two values val and val' such that $s_1 \xrightarrow{\texttt{write}\ va\ val, o_v} s_2$ and $s_1' \xrightarrow{\texttt{write}\ va'\ val', o_v} s_2'$. Since $\mathsf{eff}(\texttt{write}\ va\ val) = \mathsf{eff}(\texttt{write}\ va'\ val')$, the virtual addresses used must be the same; so $va = va'$.

Since the execution of the action is valid in both states, there must be some ma_w and ma_w' such that in the respective current page tables of o_v pt_pg and pt_pg', $pt\_pg[va] = ma_w$ and $pt\_pg'[va] = ma_w'$. In addition, we must necessarily have $get\_page\_ma(s_1, ma_w) = pg_w$ and

$get\_page\_ma(s_1', ma_w') = pg_w'$, where pg_w and pg_w' are readable/writable. Since va must be accessible, by condition 4 of valid state these pages must also be owned by the victim.

To prove the equivalence for attacker pages, we assume that $get\_page\_hyp(s_2, o_a, pa) = (ma, pg)$, and have to prove $get\_page\_hyp(s_2', o_a, pa) = (ma', pg)$. The valid state invariant lemma tells us that s_2 and s_2' are valid. Therefore, pg must be owned by o_a. Furthermore, since neither the hypervisor mapping or the page tables change during the execution of the writes, the page pg_w and pg_w' are owned by o_v in s_2 and s_2' respectively. It is easy to see that the only page that (possibly) changes in s_1 after the write is pg_w and it is not in ma, since the owners are different. Therefore $get\_page\_hyp(s_1, o_a, pa) = get\_page\_hyp(s_2, o_a, pa) = (ma, pg)$. Since s_1 and s_1' are equivalent, it must be the case that $get\_page\_hyp(s_1', o_a, pa) = (ma', pg)$. Again, it can be shown that pg does not change during the execution of the action in s_1', and therefore $get\_page\_hyp(s_2', o_a, pa) = (ma', pg)$.

For the remaining case we have to show that if $get\_page\_hyp(s_2, o_v, pa) = (ma, pg)$, then there is a ma' and pg' in s_2' such that $get\_page\_hyp(s_2', o_v, pa) = (ma', pg')$ where pg and pg' are equal except in their contents. Using condition 6 of valid state, we get a physical address pa_w such that $get\_page\_hyp(s_1, o_v, pa_w) = (ma_w, pg_w)$. By the equivalence of cache and memory mappings between s_1 and s_1', we know that $get\_page\_hyp(s_1', o_v, pa_w) = (ma_w', pg_w')$.

Now we must consider two cases, when pa is equal to pa_w and when they differ. If they are equal, then the pages in pa are equal to pg_w and pg_w' with the only difference of their value, which are val and val' respectively. If the physical addresses are different, the pages did not change, and by the equivalence of s_1 and s_1', they must be equal except for the value. This concludes the proof of the equivalence of hypervisor mapping, for the victim `write` action. □

As is typical of a non-leakage result, the other lemma generally used in non-interference, the locally preserves unwinding lemma, is not necessary. A lemma of this kind would state that the effects of some action are not observable by the attacker: if $s_1 \xrightarrow{a,o} s_2$, then $s_1 \sim s_2$, for some suitable action a.

Indeed, we do not need this result for our (current) purposes since, as we are modeling constant-time victims, we require that the victim performs the same number of actions in the same order in both executions[1]. This means that the attacker and the victim executions will be in lockstep, and therefore only lemmas that advance by one action in each execution are necessary.

Note that extra actions in one of the executions represent a victim unbalanced branch that depends on secret data. This branching is not allowed in constant-time implementations, so our requirements for the victim are consistent with this hypothesis. This will be made clearer in Chapter 4, where we define a static analysis on victim code to guarantee that the implementation is constant-time.

3.6 Isolation Theorem

The isolation theorem is one of our main results, putting together our unwinding lemmas and the interleaving of two traces of victim actions. In order to capture the condition that the victim actions executed in both traces have the same effects, we define the $\approx$ relation as follows:

$$\frac{\mathsf{eff}(a) = \mathsf{eff}(a') \quad v\_str \approx v\_str'}{(a :: v\_str) \approx (a' :: v\_str')}$$

[1]This lemma will be necessary when we introduce Stealth Memory in Chapter 5, where the isolation theorem corresponds to a non-influence property.

The theorem states that given two initial and equivalent states, and two traces of victim actions with the same effects, the attacker cannot distinguish between the two execution traces generated by interleaving the traces of victim actions:

Theorem 3.6.1 (Constant-time isolation)**.** Let s_0 and s'_0 be two states such that $s_0 \sim s'_0$; let v_str and v_str' two sequence of victim actions such that $v\_str \approx v\_str'$; and let Θ and Θ' be traces such that $interleave(s_0, [], v\_str, \Theta)$ and $interleave(s'_0, [], v\_str', \Theta')$. Then $\Theta \sim \Theta'$.

Proof. The proof is done by co-induction, using the unwinding lemmas where appropriate. Intuitively, partial traces are always equivalent, so the attacker will at each point select the next victim action (in which case they must have the same effects) or the same action for himself. This guarantees that both interleaved traces are in lockstep, so we can apply the unwinding lemmas to finally prove the equivalence of the traces. □

Note that the condition that victim actions have the same effect in both traces is necessary because the execution of two actions with different effects, a `read` to va in Θ and a `read` to va' in Θ' for instance, may provoke different attacker entries to be evicted from the cache, leading to different observations for the attacker. An access to different addresses in the traces represents a victim that makes memory accesses that depend on secret data (as in a table look-up, with the key as the index). This kind of access is forbidden in constant-time implementations.

Furthermore, this is a condition that depends exclusively on the victim behavior. This will allow us to connect this result with an application level analysis, to give formal guarantees of non leakage for constant-time programs (written in the C language). The next chapter explains this connection in detail.

On the other hand, we can also prove a result on the attacker observations, in the style of *physically observable cryptography*. We show that the attacker cannot gain information by calling the leakage oracle after the execution of an action:

Lemma 3.6.2 (Step non-leakage for equivalent states)**.** Let s_1 and s'_1 be states such that $s_1 \xrightarrow{a,o} s_2$, $s'_1 \xrightarrow{a',o} s'_2$ and $s_1 \sim s'_1$. Furthermore, a and a' are the same action if o is the attacker, or have the same effect if it is the victim. Then, $\ell(s_1, s_2) = \ell(s'_1, s'_2)$.

Proof. The proof is based on the equivalence of s_1 and s'_1 to show that after the execution of the actions, the same cache entries are evicted from both states. The equivalence relation has to be strong enough for this to be the case, and this is the main reason our equivalence is so complex, involving many state components and different conditions, as a simpler relation would not be enough to guarantee that the attacker observations are the same. This strong equivalence makes the proof of the unwinding lemmas and the main isolation theorem much more complex, as these results are correspondingly stronger. □

This lemma can be easily generalized to show that the leakage oracle returns the same observations for each state in two equivalent traces: if $\Theta \sim \Theta'$, then for all i, $\ell(\Theta[i].st, \Theta[i{+}1].st) = \ell(\Theta'[i], \Theta'[i+1])$.

Since the scheduler model is very general and can be influenced by the attacker, we are able to obtain a very strong isolation result. However, the approach we took still has some shortcomings. The scheduler is based on discrete actions, and not based on timing, as it is usually the case (i.e. instead of giving each OS the same time slice to execute, our example round robin scheduler executes the same number of actions for each OS). Modeling a more realistic scheduler would pose a significant challenge, since it would require to have a concrete timing model for the target architecture instructions. This kind of analysis would be extremely difficult in the context of program verification, where it would be necessary to guarantee that the timing of the execution of the victim is independent of secret values.

Chapter 4

Language-based security

In the previous chapter we presented an isolation result that guarantees non-leakage on the idealized virtualization model, under the assumption that the addresses accessed by the victim process were independent of its data. As mentioned at that time, this condition only depends on the behavior of the victim, and can be checked on the victim program source code. In this chapter we describe a way of guaranteeing non-leakage at the application level, by performing a static analysis on programs. The analysis enforces that the program is constant-time, i.e. that the sequence of memory accesses is independent of the value of some secret information that needs to be protected against leakage. We will show that constant-time programs validated by our analysis, are indeed protected against cache side-channel attacks when running together with a malicious operating system in a virtualization platforms, by linking the static analysis at the application level with the isolation results at the (idealized) system level.

Our analysis is built on top of CompCert [93], a formally verified, optimizing C compiler that generates reasonably efficient assembly code for x86 platforms (as well as PowerPC and ARM). In addition to being a significant achievement on its own, CompCert provides an excellent platform for developing verified static analyses. We take specific advantage of two features of CompCert:

1. its memory model, which achieves a subtle and effective compromise between exposure to machine-level representation of memory and tractability of formal proofs, and is ideal for reasoning about properties that relate to sequences of memory accesses;

2. its sophisticated compilation chain, which involves over 15 passes, and about 10 intermediate languages, which are judiciously chosen to provide compact representations on which program analyses can be verified.

Our goal is to implement a static data-flow analysis for checking whether programs perform conditional jumps or memory accesses that depend on secrets, and to derive strong semantical guarantees for the class of programs accepted by the analysis. In Section 4.1 we give some background on data-flow analyses in general.

In order to obtain meaningful results, it is important that our analysis is performed on intermediate representations towards the end of the compilation chain, rather than source C programs; indeed, some compilation passes in the compiler middle-end (typically at RTL level) may modify and reorder memory accesses and hence a non-leaking C program could well be transformed into a leaking x86 program, or vice versa. This happens regardless of the semantic preservation property guaranteed by CompCert, since it does not guarantee that the sequence of memory accesses of the program will remain unchanged. Indeed, some compiler passes such as register allocation, will indeed modify the sequence of memory accesses of the program. We will therefore settle on performing our analysis on a minor variation of the nearly final intermediate

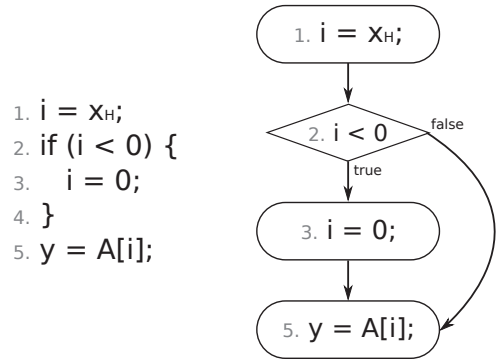

Figure 4.1: Example program and its CFG

language, which we called MachIR. Studying ways of performing the analysis at the source level, and providing guarantees even in the presence of some optimizations is left as future work, as this would greatly simplify the static analysis.

In Section 4.2 the syntax and semantics of the MachIR intermediate language is presented. In Section 4.3, we show that the sequence of memory addresses of a program in Mach, MachIR and assembly are equal, in order to be guarantee that the properties we prove also hold for the program executable code. Sections 4.4 and 4.5 present two related static analysis. The first computes the memory addresses that will be accessed at run-time by the program, while the second type verifies that the program is constant-time, by means of a type system that guarantees that memory accesses do not depend on secret information.

Our first result on programs that are accepted by the type system is a soundness lemma presented in Section 4.6. Intuitively, this guarantees that programs that type-check behave as required by the isolation theorem presented in Chapter 3, i.e. that all memory addresses accessed by the program are independent of secret information. The formal link between the two models is described in Section 4.7. Finally, Section 4.8 concludes the chapter with a discussion of limitations and future work related to our constant-time type system.

4.1 Data-flow analysis

Data-flow analysis is a technique used to deduce information about the run-time behavior of a program, by only studying the static program code [79] Its most common applications are in implementing compiler optimizations, such as constant propagation and liveness analysis; and in information flow analysis [52]. Here we will use it to enforce a constant-time behavior on programs, and therefore guarantee that secrets do not leak through cache side-channel attacks.

Data-flow analyses usually work on a program representation called the *control-flow graph* or CFG. In a CFG each node corresponds to a program instruction, and there is a transition between each node and the node or nodes corresponding to the next instructions in the code. Figure 4.1 shows an example program and its CFG representation.

The goal of the analysis is to compute, for each instruction, facts that are guaranteed to hold in every possible execution of the program. For this, one defines what information is present at the start of the program, how the execution of each node changes this information, and how to combine the information when a node has more than one incoming edges. More formally, an instance of a data-flow problem (DFP) includes a CFG, a domain D of data-flow facts, an initial fact $init$, an operator $\sqcap$ that combines incoming facts, and a transfer function $f_n : D \to D$ that computes the effect of executing n.

Kildall [79] studied this kind of problems and proposed an algorithm to compute the fixpoint

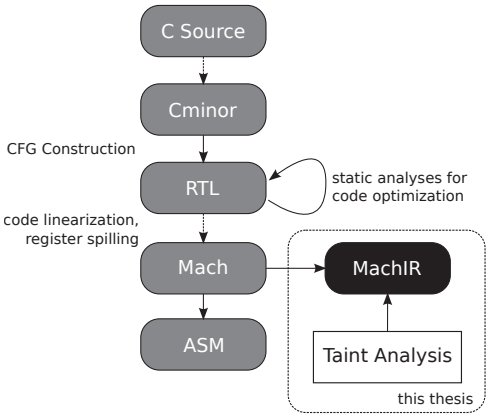

Figure 4.2: CompCert architecture overview with MachIR addition

for the DFP, that always finds the best solution (*meet over all paths* or MOP), provided the domain D is a complete lattice with no infinite descending chains, $\sqcap$ is the lattice meet operator, and f_n is distributive.

Data-flow analysis, and Kildall's algorithm in particular, can be used to formally study a program information flow behavior. Secure information flow analysis shows how information travels between variables in the program, detecting undesirable flows. Usually one classifies program variables with a security level (generally just public or *Low*, and secret of *High*), and forbids the assignment of a High value to a Low variable, through either explicit (called flow-insensitive information flow) or also implicit assignments (called flow-sensitive).

In this work, instead of forbidding flow from High to Low variables, we will *taint* or mark as High variables that receive the value of a High variable. We will then check that a program never uses a High variable in the condition of a branch or loop, or as an array index. The example program of Figure 4.1, for example, would be rejected by our analysis assuming x_H is High. Indeed, the variable i is tainted at instruction 1, and it is used in the condition of the if at instruction 2, which is not allowed. In addition, at instruction 5, the same High variable i is used to access array A, which is also forbidden.

4.2 MachIR syntax and semantics

The static analysis we are interested in requires reasoning about the sequence of memory accesses of the program. For this kind of analysis, the most appropriate representation is Mach, the last-but-final intermediate language in the CompCert compilation chain. This language is used after compiler passes that may introduce new memory accesses, and immediately before generation of assembly code. Hence the sequence of memory accesses at Mach and assembly levels coincide. Moreover, Mach has a compact syntax, which is important to reduce proof effort.

On the other hand, the Mach language does not enjoy a control flow graph representation, which is a drawback for performing static analyses. We therefore adopt a minor variant of Mach, which we call MachIR, that retains the same instruction set as Mach but makes explicit the successor(s) of each instruction, see Figure 4.2. MachIR is a suitable representation for building verified static analyses about sequences of memory accesses of programs.

A MachIR program p is represented by a (partial) map of program nodes to instructions, i.e. as an element of $\mathbb{N} \rightharpoonup \mathbb{I}$. Each instruction explicitly carries its successor(s) node(s) in the control flow graph.

The most basic instructions manipulate registers and perform conditional and unconditional

$\mathbb{N} \ni$	n		CFG nodes
$\mathbb{R} \ni$	r		register names
$\mathbb{S} \ni$	S		global variable names
$\mathbb{A} \ni addr$	::=		
	|	$\mathsf{based}(\mathcal{S})$	based addressing
	|	$\mathsf{stack}(\delta)$	stack position
	|	$\mathsf{indexed}$	indexed addressing
$\mathbb{O} \ni op$	::=		
	|	$\mathsf{addrof}(addr)$	symbol address
	|	move	register move
	|	$\mathsf{arith}(a)$	arithmetic operation
$\mathbb{I} \ni instr$	::=		
	|	$\mathsf{op}(op, \vec{r}, r, n)$	register operation
	|	$\mathsf{load}_\varsigma(addr, \vec{r}, r, n)$	memory load
	|	$\mathsf{store}_\varsigma(addr, \vec{r}, r, n)$	memory store
	|	$\mathsf{goto}(n)$	static jump
	|	$\mathsf{cond}(c, \vec{r}, n_{then}, n_{else})$	conditional static jump

Figure 4.3: Instruction set

jumps: $\mathsf{op}(op, \vec{r}, r, n)$ (register r is assigned the result of the operation op on arguments $\vec{r}$; next node is n), $\mathsf{goto}(n)$ (unconditional jump to node n) and $\mathsf{cond}(c, \vec{r}, n_{then}, n_{else})$ (conditional jump; next node is n_{then} or n_{else} depending on the boolean value that is obtained by evaluating condition c on arguments $\vec{r}$).

Memory is manipulated trough two instructions: $\mathsf{load}_\varsigma(addr, \vec{r}, r, n)$ (register r receives the content of the memory at an address that is computed with addressing mode $addr$ and arguments $\vec{r}$; next node is n) and $\mathsf{store}_\varsigma(addr, \vec{r}, r, n)$ (the content of the register r is stored in memory at an address that is computed with addressing mode $addr$ and arguments $\vec{r}$; next node is n). ς describes the type of memory chunk that is accessed (of size 1, 2 or 4 bytes).

Addressing $\mathsf{based}(\mathcal{S})$ (resp. $\mathsf{stack}(\delta)$) directly denotes the address of a global symbol (resp. of the stack memory block). Pointer arithmetic is performed through addressing mode $\mathsf{indexed}$. Additional instructions are used to access the activation record of a procedure call, and to perform the call. Figure 4.3 gives an excerpt of the language instruction set.

Figure 4.4 presents part of the semantics of the MachIR language. Values are either numeric values $\mathsf{Vnum}(i)$ or pointer values $\mathsf{Vptr}(b, \delta)$ with b a memory block name and δ a block offset. We let $\&\mathsf{SP}$ denote the memory block that stores the stack. A state $t = (n, \rho, \mu)$[1] is composed of the current CFG node n, the set of register values ρ and the $\mathsf{CompCert}$ memory μ.

The operational semantics is modelled with judgments:

$$t \xhookrightarrow{e} t'$$

The semantics is implicitly parameterized by a program p. Informally, the judgment above says that executing the program p with state t leads to a state t', and has visible effect e, where e is either a read effect $\mathsf{read}\ x$ (with x an address), or a write effect $\mathsf{write}\ x$, or the null effect $\emptyset$. Note that effects model the addresses that are read and written, but not their value. Figure 4.5

[1] To avoid confusion, we will use the letter t to denote states at the language level, and s to denote states at the virtualization platform level.

$$
\begin{array}{rll}
\mathbf{Num} \ni i & & \text{numerics} \\
\mathbf{Block} \ni b & & \text{memory block identifiers} \\
\mathbf{Val} \ni v & ::= & \\
& \mid \mathsf{Vnum}(i) & \text{numeric value} \\
& \mid \mathsf{Vptr}(b, i) & \text{pointer value (block+offset)} \\
\\
\mathbb{S} \to \mathbf{Block} \ni \& & & \text{address of symbols} \\
\mathbb{R} \to \mathbf{Val} \ni \rho & & \text{set of register values} \\
\mathbf{Mem} \ni \mu & & \text{CompCert memory}
\end{array}
$$

Figure 4.4: Semantic domains

$$
\begin{array}{rcl}
[\![\mathsf{indexed}]\!](\rho, [r_1; r_2]) = & [\![+]\!](\rho(r_1), \rho(r_2)) \\
[\![\mathsf{global}(S)]\!](\rho, []) & = & \mathsf{Vptr}(\&S, 0) \\
[\![\mathsf{stack}(\delta)]\!](\rho, []) & = & \mathsf{Vptr}(\&SP, \delta) \\
& = & \text{undef otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\mathsf{addrof}(addr)]\!](\rho, \vec{r}) & = & [\![addr]\!](\rho, \vec{r}) \\
[\![\mathsf{move}]\!](\rho, [r]) & = & \rho(r) \\
[\![\mathsf{arith}(a)]\!](\rho, \vec{r}) & = & [\![a]\!](\rho[\vec{r}])
\end{array}
$$

$$
\frac{p[n] = \mathsf{op}(op, \vec{r}, r, n')}{(n, \rho, \mu) \xrightarrow{\emptyset}_P (n', \rho[r \mapsto [\![op]\!](\rho, \vec{r})], \mu)}
$$

$$
\frac{p[n] = \mathsf{load}_\varsigma(addr, \vec{r}, r, n') \qquad [\![addr]\!](\rho, \vec{r}) = v_{\text{addr}} \qquad \mu[v_{\text{addr}}]_\varsigma = val}{(n, \rho, \mu) \xrightarrow{\texttt{read } v_{\text{addr}}}_P (n', \rho[r \mapsto val], \mu)}
$$

$$
\frac{p[n] = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \qquad [\![addr]\!](\rho, \vec{r}) = v_{\text{addr}} \qquad store(\mu, \varsigma, v_{\text{addr}}, \rho(r)) = \mu'}{(n, \rho, \mu) \xrightarrow{\texttt{write } v_{\text{addr}}}_P (n', \rho, \mu')}
$$

Figure 4.5: Mach IR semantics (excerpt)

presents selected rules of the semantics. Note that an instruction like $\mathsf{store}_4(\mathsf{stack}(\delta), [], r, n')$ will assign the four stack positions δ, $\delta + 1$, $\delta + 2$ and $\delta + 3$.

4.3 Equivalence between MachIR and assembly

In order to be able to perform our analysis using the MachIR language and guarantee our results are valid in the resulting object program, it is necessary to show that it is sound and complete to enforce standard, path, and memory-trace non-interference based on the MachIR (rather than the ASM) representation of programs. Indeed, note that all languages in CompCert, including ASM, Mach and MachIR share a same notion of initial state where each global variable is given an initial value. We prove that for every Mach program p, the execution traces of p, its MachIR form p_{MachIR}, and its ASM form p_{ASM} are equal.

4.4 Alias analysis

In order to enforce non-interference policies, constant-time verification must track how information flows during execution. However, instructions such as $\mathsf{store}_\varsigma(\mathsf{indexed}, [r_1; r_2], r, n')$ do not show explicitly which memory cell will be assigned at runtime. The standard solution, used e.g. by Robert and Leroy [94] for the CompCert RTL language, is to perform an alias analysis. For our purposes, it is sufficient to perform a simple type-based analysis, which is similar to a points-to analysis [71] but restricts the set of points-to information to a singleton for each variable.

Our type system statically computes the points-to information $\mathit{PointsTo}(n, \mathit{addr}, \vec{r})$ at every node n for a memory access with an addressing mode addr and arguments $\vec{r}$. Hence, if node n contains an instruction $\mathsf{load}_\varsigma(\mathit{addr}, \vec{r}, r, n')$ or an instruction $\mathsf{store}_\varsigma(\mathit{addr}, \vec{r}, r, n')$, we have a prediction, at compile time, of the targeted memory address.

We first define the set alias of alias information. An element of alias is of one of the forms:

1. Num, which represents only a numerical value;
2. $\mathsf{Symb}(\mathcal{S})$, which represents either a non-pointer value, or a pointer value that points to the variable $\mathcal{S}$;
3. $\mathsf{Stack}(\delta)$, which represents either a non-pointer value, or $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$.

For example, if an instruction $\mathsf{store}_\varsigma(\mathsf{indexed}, [r_1; r_2], r, n')$ is performed at node n when r_1 contains $\mathsf{Vptr}(\&\mathcal{S}, 8)$ and r_2 contains the integer 16, the points-to static analysis may safely predict $\mathit{PointsTo}(n, \mathit{addr}, \vec{r}) = \mathsf{Symb}(\mathcal{S})$, because the accessed pointer is $\mathsf{Vptr}(\&\mathcal{S}, 24)$.

Alias information is partially ordered with $\mathsf{Num} \subseteq \mathsf{Symb}(\mathcal{S})$ and $\mathsf{Num} \subseteq \mathsf{Stack}(\delta)$. This partial order is lifted to alias maps by the standard pointwise lifting.

Programs are assigned types of the form:

$$A \in (\mathbb{N} \to \mathbb{S} \to \mathit{alias}) \times (\mathbb{N} \to \mathbb{N} \to \mathit{alias}) \times (\mathbb{N} \to \mathbb{R} \to \mathit{alias})$$

where for every node n:

- $A[n](S)$ gives the flow-sensitive points-to information of the global variable S;
- $A[n](\delta)$ gives the flow-sensitive information of the stack cell at address $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$;
- $A[n](r)$ gives the flow-sensitive information of the register r.

The typing rules are given in Figure 4.6, and are mostly standard. Each load and store instruction requires two cases: one for stack access and one other for global variable access. For a load at node n, in a register r and on a pointer value that points to a global variable $\mathcal{S}$, we transfer the abstract content of $A[n](\mathcal{S})$ to the abstract information $A[n'](r)$ of the successor node n'. The treatment is similar for a load on a pointer value that points to a stack position. For a store at node n, from a register r to a pointer value that points to a stack position $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$, we update the abstract content $A[n'](\delta)$ of the successor node n'. If the store is performed on a global variable, we only perform a weak update: $A[n'](\mathcal{S})$ must contain $A[n](r)$ (the content of the stored value), but also $A[n](\mathcal{S})$ (the previous content), since the $\mathsf{Symb}(\mathcal{S})$ may represent several concrete memory cells. The type system will reject some programs if a pointer value, at a specific program point, may points-to a different symbol, depending on the path that has been followed to reach this point. This restriction could be lifted by considering sets (rather than singletons) of symbolic addresses.

$$
\begin{array}{lll}
\mathit{alias} & ::= & \\
& | \quad \mathsf{Num} & \text{numerical value} \\
& | \quad \mathsf{Symb}(\mathcal{S}) & \text{points to any cell allocated for symbol } \mathcal{S} \\
& | \quad \mathsf{Stack}(\delta) & \text{points to the } \delta^{\text{th}} \text{ stack cell}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}[\![\mathsf{indexed}]\!](a,[r_1;r_2]) & = & \mathcal{A}[\![+]\!]([a(r_1);a(r_2)]) \\
\mathcal{A}[\![\mathsf{global}(S)]\!](a,\vec{r}) & = & \mathsf{Symb}(S) \\
\mathcal{A}[\![\mathsf{stack}(\delta)]\!](a,[]) & = & \mathsf{Stack}(\delta) \\
& = & \mathsf{Num} \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{A}[\![\mathsf{addrof}(\mathit{addr})]\!](a,\vec{r}) & = & \mathcal{A}[\![\mathit{addr}]\!](a,\vec{r}) \\
\mathcal{A}[\![\mathsf{move}]\!](a,[r]) & = & a(r) \\
\mathcal{A}[\![\mathsf{arith}(a)]\!](a,\vec{r}) & = & \mathcal{A}[\![a]\!](a[\vec{r}])
\end{array}
$$

$$
\frac{A[n][r \mapsto \mathcal{A}[\![\mathit{op}]\!](A[n],\vec{r})] \subseteq A[n']}{A \vdash n : \mathsf{op}(\mathit{op},\vec{r},r,n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n],\vec{r}) = \mathsf{Symb}(S) \qquad A[n][r \mapsto A[n](S)] \subseteq A[n']}{A \vdash n : \mathsf{load}_\varsigma(\mathit{addr},\vec{r},r,n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n],\vec{r}) = \mathsf{Stack}(\delta)\} \qquad A[n][r \mapsto A[n](\delta)] \subseteq A[n']}{A \vdash n : \mathsf{load}_\varsigma(\mathit{addr},\vec{r},r,n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n],\vec{r}) = \mathsf{Symb}(S) \qquad A[n][S \mapsto A[n](r) \cup A[n](S)] \subseteq A[n']}{A \vdash n : \mathsf{store}_\varsigma(\mathit{addr},\vec{r},r,n')}
$$

$$
\frac{\mathcal{A}[\![\mathit{addr}]\!](A[n],\vec{r}) = \mathsf{Stack}(\delta) \qquad A[n][\delta \mapsto A[n](r)] \subseteq A[n']}{A \vdash n : \mathsf{store}_\varsigma(\mathit{addr},\vec{r},r,n')}
$$

$$
\frac{A[n] \subseteq A[n']}{A \vdash n : \mathsf{goto}(n')} \qquad \frac{A[n] \subseteq A[n_{then}] \quad A[n] \subseteq A[n_{else}]}{A \vdash n : \mathsf{cond}(c,\vec{r},n_{then},n_{else})}
$$

Figure 4.6: Alias type system

$$\frac{p(n) = \mathsf{op}(op, \vec{r}, r, n')}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\vec{r})]}$$

$$\frac{p(n) = \mathsf{load}_\varsigma(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \qquad \tau(\vec{r}) = \mathsf{Low}}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto X_h(\mathcal{S})]}$$

$$\frac{p(n) = \mathsf{load}_\varsigma(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathsf{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\delta) \sqcup \cdots \sqcup \tau(\delta + \varsigma - 1)]}$$

$$\frac{\begin{array}{c} p(n) = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \\ PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \qquad \tau(\vec{r}) = \mathsf{Low} \qquad \tau(r) \sqsubseteq X_h(\mathcal{S}) \end{array}}{X_h \vdash n : \tau \Rightarrow \tau}$$

$$\frac{p(n) = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathsf{Stack}(\delta)}{X_h \vdash n : \tau \Rightarrow \tau[\delta \mapsto \tau(r), \ldots, \delta + \varsigma - 1 \mapsto \tau(r)]}$$

$$\frac{p(n) = \mathsf{goto}(n')}{X_h \vdash n : \tau \Rightarrow \tau}$$

Figure 4.7: Information flow rules for constant-time

Definition 4.4.1 (Alias-well-typed programs)**.** A program p is alias-well-typed with respect to an alias map A, if $A \vdash n : p[n]$ holds for all nodes n in p, and at the initial node n_0 of the program, $A[n_0]$ contains a correct abstraction of the initial program memory.

We forge the initial abstraction of the memory by scanning the declaration of the global variables. If a global is initialised with a numeric constant, its initial abstraction is Num. CompCert also allows to initialise a global with the address of an other global variable $\mathcal{S}$. In this case, the initial abstraction is $\mathsf{Symb}(\mathcal{S})$.

We note that the relative simplicity of our analysis is derived from the specific guarantees provided by the CompCert memory model, and more particularly from the strong isolation between global variables. The main complexity of our type system lies in its treatment of stack manipulation, which requires specific care because the CompCert memory model does not enforce any separation guarantee for it. The type system makes sure every access to the stack is performed through constant addresses in order to track finely this part of the memory.

4.5 Constant-time type system

We define a constant-time type system that enforces standard, path and memory-trace non-interference of programs. Informally, the type system prevents branching on secrets and performing memory accesses that depends on a secret – otherwise the access would leak information.

As usual, we consider a lattice of security levels $\mathbb{L} = \{\mathsf{Low}, \mathsf{High}\}$ with $\mathsf{Low} \sqsubseteq \mathsf{High}$. Initially, the user declares a set $X_h^0 \subseteq \mathbb{S}$ of high variables.

Programs are assigned types (X_h, T), where $X_h \in \mathbb{S} \to \mathbb{L}$ is a global type, and $T \in \mathbb{N} \to (\mathbb{N} + \mathbb{R}) \to \mathbb{L}$ is a mapping from program nodes to local types. X_h is a flow-insensitive global type which assigns a security level $X_h(\mathcal{S})$ for every global variable $\mathcal{S} \in \mathbb{S}$. T is a flow-sensitive local type which assigns for every offset $\delta \in \mathbb{N}$ the security level $T[n](\delta)$ of the stack cell at address $\mathsf{Vptr}(\&\mathsf{SP}, \delta)$ and node n, and for every register $r \in \mathbb{R}$ its security level $T[n](r)$ at node n. Formally, the type system manipulates judgments of the form:

$$X_h \vdash n : \tau_1 \Rightarrow \tau_2$$

where X_h is a global type, n is a node, and τ_1 and τ_2 are local types, i.e. $\tau_1, \tau_2 \in (\mathbb{N} + \mathbb{R}) \to \mathbb{L}$. The type system enforces a set of constraints on X_h^0, X_h and T. Typing rules are given in Figure 4.7; we note $\tau(\vec{r})$ for $\bigsqcup_{r \in \vec{r}} \tau(r)$.

The rule for $\mathsf{op}(op, \vec{r}, r, n')$ simply updates the security level of r with the supremum of the security levels of $\vec{r}$.

There are two rules for $\mathsf{load}_\varsigma(addr, \vec{r}, r, n')$. The first one considers the case where the value is loaded from a global variable $\mathcal{S}$. In this case, the typing rule requires that all registers are low, i.e. $\tau(\vec{r}) = \mathsf{Low}$, as we want to forbid memory accesses that depend on a secret. The security level of the register r is updated with the security level $X_h(\mathcal{S})$ of the variable. The second rule considers the case where the value is loaded from a stack position at offset δ. In this case, our type system conservatively requires that the memory access is constant (and statically predicted by the alias type system). Note that the security level of the register r is set to the maximum of $\tau(\delta), \ldots, \tau(\delta + \varsigma - 1)$. Indeed, the security level of $\tau(\delta)$ models the level of the 8-bit value at position δ; if the load is performed with a memory chunk of size strictly bigger than 1, several 8-bit value will be accessed. Our type system takes care of this subtlety.

The two typing rules for store are similar to the rules for load. If the store is performed on a global variable, we again require $\tau(\vec{r}) = \mathsf{Low}$ to make sure the dereferenced pointer address does not depend on secrets. The constraint $\tau(r) \subseteq X_h(\mathcal{S})$ propagates the security level of the stored value. For a store on a stack offset, we again make sure to consider enough stack offsets by considering the memory chunk of the instruction.

Definition 4.5.1 (Constant-time programs)**.**
A program p is constant-time with respect to a set of variables X_h^0, written $X_h^0 \vdash p$, if there exists (X_h, T) such that for every $\mathcal{S} \in X_h^0$, $X_h(\mathcal{S}) = \mathsf{High}$ and for all nodes n and all its successors n', there exists τ such that

$$X_h \vdash n : T(n) \Rightarrow \tau \quad \wedge \quad \tau \sqsubseteq T(n')$$

where $\sqsubseteq$ is the natural lifting of $\sqsubseteq$ from $\mathbb{L}$ to types.

To type-check a program, the user needs to specify the set X_h^0 of initial secrets variables to protect, then our compiler computes the sets X_h and T using Kildall's algorithm [79], and finally verifies if the program is constant-time, according to Definition 4.5.1.

4.6 Soundness of type system

The soundness property of our type system shows that programs that type-check are constant-time. This means that, assuming all Low variables are equal in two initial states with arbitrary High variables, the victim performs the same sequence of memory accesses. If this is the case, the values of the (arbitrary) High variables have no effect on the addresses accessed by the victim, and it is therefore constant-time.

We define an indistinguishability relation between states, noted $t \sim_{X_h} t'$, to capture the requirement that all Low values (either global variables, stack cells or registers) are equal in t and t'. Figure 4.8 shows the formal definition of this relation. We can then state our soundness result:

Theorem 4.6.1 (Soundness of constant-time verification)**.** Let p be a program that is alias well-typed and constant-time well-typed with respect to a set X_h of high variables. For every two executions of p:

$$t_1 \xrightarrow{e_1} t_2 \xrightarrow{e_2} t_3 \xrightarrow{e_3} \ldots \qquad\qquad t'_1 \xrightarrow{e'_1} t'_2 \xrightarrow{e'_2} t'_3 \xrightarrow{e'_3} \ldots$$

if $t_1 \sim_{X_h} t'_1$, then for all i, $t_i \sim_{X_h} t'_i$ and $e_i = e'_i$.

$$\frac{\&S = b \qquad X_h(S) = \mathsf{High}}{HighPtr^n_{X_h}(b,i)} \qquad \frac{X_h[n](\delta) = \mathsf{High}}{HighPtr^n_{X_h}(\&\mathsf{SP},\delta)}$$

$$\frac{\forall b, \forall i, \forall \varsigma, \mu_1[\mathsf{Vptr}(b,i)]_\varsigma = \mu_2[\mathsf{Vptr}(b,i)]_\varsigma \vee HighPtr^n_{X_h}(b,i)}{\mu_1 \sim^n_{X_h} \mu_2}$$

$$\frac{\forall r,\ X_h[n](r) = \mathsf{Low} \implies \rho_1(r) = \rho_2(r)}{\rho_1 \sim^n_{X_h} \rho_2}$$

$$\frac{\rho_1 \sim^n_{X_h} \rho_2 \qquad \mu_1 \sim^n_{X_h} \mu_2}{(n,\rho_1,\mu_1) \sim_{X_h} (n,\rho_2,\mu_2)}$$

Figure 4.8: Indistinguishability relations

Proof. If the program type-checks, it can be shown that after the execution of two instructions with the same effect (two reads or two writes to the same addresses, or two instructions with no effect) from two indistinguishable states, the resulting states will also be indistinguishable. This is because if the instruction uses some value as a parameter, it is either High in both states and therefore the type system will mark as High all variables that get updated by this instruction; or they are equal in both states and all variables will change to the same value in the resulting states.

In order to show that instruction effects are the same in both traces we show that if there is a branch in the program, it will take the same path in both executions. This is because High branches are forbidden by the type system, and therefore the branch condition will be Low and equal in both states. In addition, expressions used as addresses in all instructions are guaranteed to be Low (and therefore equal), since the type system forbids dereferencing High pointers. This shows that the effects of program instructions are equal in both executions. □

4.7 System-level security

In this section we establish a link between the constant-time analysis at the language level and the isolation result stated in theorem 3.6.1 at the system level; in order to give formal guarantees that constant-time implementations are secure against cache timing attacks on virtualization platforms.

With the definitions in Section 3.4 and the semantics defined for MachIR programs, one can define the concurrent execution $(\mathcal{A} \parallel p)[s_0]$ of an attacker $\mathcal{A}$ and a victim program p, starting from an initial state s_0. Informally, $(\mathcal{A} \parallel p)[s_0]$ is the platform execution trace that interleaves the sequence of actions by the victim operating system o_v, corresponding to the semantics of the program p; and the sequence of actions chosen by the adversary o_a, according to the adversarially-chosen scheduling policy—both captured in the definition of $\mathcal{A}$.

Given a program p and a set of initial secrets X^0_h, we define an equivalence relation $\sim_{X^0_h}$ on platform states that requires that low variables have the same value. This relation is implicitly parameterized by a mapping of MachIR states to platform states that maps program variables to platform virtual addresses.

We can now state and prove a theorem that guarantees that if a program p is constant-time $(X^0_h \vdash p)$ with respect to the set of secrets X^0_h, then an attacker $\mathcal{A}$ cannot distinguish between two different executions of this program from two initial states that are indistinguishable and have the same Low values. Formally:

Theorem 4.7.1 (System-level security)**.** Given a program p, an attacker $\mathcal{A}$, and initial states s_0 and s'_0; such that $s_0 \sim s'_0$ and $s_0 \sim_{X_h^0} s'_0$. If $X_h^0 \vdash p$, then $(\mathcal{A} \parallel p)[s_0] \sim (\mathcal{A} \parallel p)[s'_0]$.

Proof. The proof of this theorem is a straightforward combination of theorems 4.6.1 and 3.6.1. Since we are in the hypothesis of the first, one can prove that the sequence of memory accesses of the victim are the same in both executions and therefore, by Theorem 3.6.1, the attacker cannot distinguish between the two executions. □

In order to state the this theorem, we defined an interface in our Coq development to link the virtualization model and its isolation theorems presented in previous chapters, to the static analysis and proofs presented here. For this we use the Coq module system [45] in order to obtain a language independent interface that can be linked to other languages, in addition to the CompCert intermediate language we use (called *MachIR*).

This interface consists of abstract definitions of the type L for the language; *Lstate* for states in the language semantics; a function *get_trace*, that given a program and an initial state returns the stream of effects generated by the execution of that program; and an equivalence relation $\sim_{X_h^0}$ that holds when two states differ only in the values of High variables.

We can easily instantiate this abstract definitions for MachIR, using the language syntax as L (see Figure 4.3), the MachIR states as *Lstate* (triples of the form (n, ρ, μ), see Figure 4.4), and the projection of effects of MachIR execution streams as the *get_trace* function (see Figure 4.5).

Theorem 4.7.1 provides the first formal proof that constant-time programs are secure against cache timing attacks on virtualization platforms, even in the presence of a strong, active attacker which can interleave at will the victim program execution with code of its choice.

4.8 Discussion

As a consequence of the simple alias analysis, our type system has some shortcomings that make it necessary to perform some minor modifications to the program source code.

Specifically, our alias analysis is not able to track usage of arrays in the stack, because tracking stack offsets precisely would require a complex value analysis, which is not implemented. Formalizing a value analysis is certainly possible, but left for further work.

In addition, in order to avoid reasoning about procedure calls, we use a functionality of CompCert to automatically inline all code into one function, before performing the analyses. Note that this implies that recursion cannot be used (which is usually not a problem for cryptography) and that calls to external functions must be removed before type-checking the program. Again, extending the tool to handle these cases is left for future work.

Chapter 5

Stealth Memory

In the previous chapters we formally analyzed the behavior of constant-time programs and showed that they are secure against cache-timing attacks. There are some cryptographic algorithms that are designed to be constant-time, and some that can be made constant-time by modifying its implementations. There are cases where efficient constant-time implementations exist, see [77] for example, where Käsper and Schwabe show an efficient constant-time implementation for AES. However these implementations are dependent of the algorithm used. In general, naively removing look-up tables may degrade performance, since it is necessary to compute at run-time the costly operations stored in memory.

In order to avoid this problem altogether, we will study another countermeasure against cache based side-channels, called *Stealth Memory*. In [57], Erlingsson and Abadi discuss the basis of this system-level protection mechanism that runs on commodity hardware and uses a special memory region intended to allocate data whose usage pattern needs to be protected. The authors claim that information stored in a stealth region is not exposed to attackers via the usual cache side-channels. An implementation of this mechanism for the Microsoft Hyper-V hypervisor, called StealthMem, is presented and discussed in [80]. In StealthMem, the system hypervisor provides each application running on a virtual machine with a limited amount of stealth memory than can be accessed as if caches were not shared with other VMs. Stealth memory can be used, for instance, by programmers of cryptographic libraries to store data, such as look-up tables, that are known to be target of cache side-channel attacks.

The solution embodies a software method for locking memory regions into the shared cache. This is accomplished by selecting a cache line set for stealth use, and disallowing the allocation of addresses that are mapped to this cache line set. This guarantees that cache entries from those regions can not be evicted by other VMs, and that accesses to those regions do not evict cache entries of other VMs. In addition, a set of cache lines is assigned to each CPU core, where different processor cores might be running different VMs. Whenever a VM is scheduled to execute, the hypervisor ensures that the VM's stealth region is loaded into the locked cache lines of the current core. Therefore each VM can use its own stealth region to store data without revealing its usage patterns.

Figure 5.1 illustrates the cache and memory setup of stealth memory on a four core machine. There is a locked page per core in the cache, in which running operating systems store their stealth memory page. On context switch, for example when VM_5 is scheduled to execute on Core 4, the stealth page of the new operating system Pg_5 is stored in the corresponding cache entry for the core.

Since addresses mapped to the stealth cache line set cannot be used by the guest operating systems or the hypervisor, rendering part of the memory unusable, the stealth memory mechanism has some storage overhead. This overhead was minimized in the original StealthMem work by

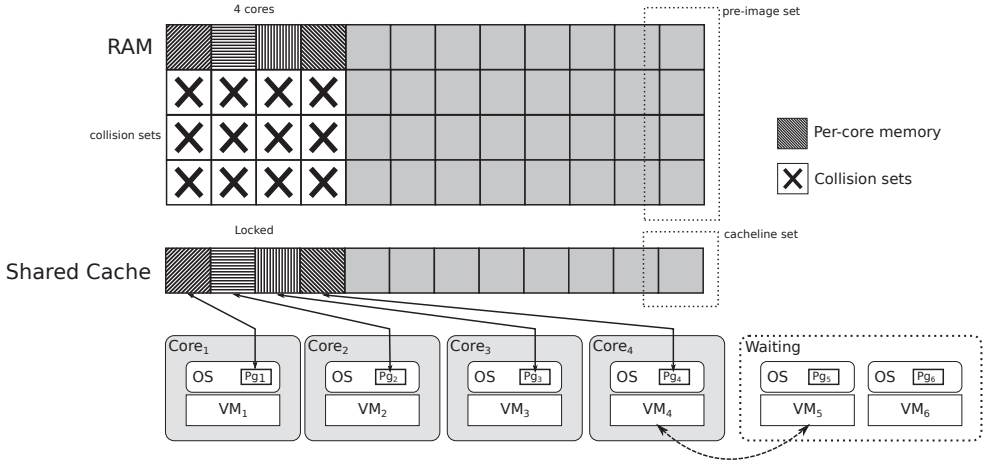

Figure 5.1: Stealth memory on multicore [80]

exploiting an architectural property of caches (set associativity) and the cache replacement policy (pseudo-LRU) present in some commodity hardware. Since we work with an abstract replacement policy, we will leave the study of this StealthMem optimization for future work.

In [80], the authors illustrate the simplicity of use of StealthMem by describing the changes that were required to transform standard implementations of Blowfish, AES and DES to use the stealth memory regions. The modifications in all cases consisted in replacing the global array variables that hold the encryption look-up tables by pointers to the stealth region. In the case of Blowfish this change required only 3 lines and DES required to change 5 lines. Adapting AES was less straightforward, it required a change of 34 lines, due to the fact that the implementation declares its tables as 8 different variables.

In order to provide convincing guarantees of security, it is necessary to deploy effective countermeasures against attacks, and to acquire expertise on their usage. Stealth memory is a relatively new development that still has shortcomings in these areas: there is no formal guarantee that information stored in stealth regions is indeed protected against side-channels, and application code needs to be manually modified to allocate some of the program data in stealth memory. There is no mechanism to correctly select which variables should be stealth, or a way to prove that a certain variable selection is safe.

In this chapter we tackle these issues by first weakening the definition of constant-time, introducing a new program classification which we call *S-constant-time*, that captures the behavior of programs that correctly use stealth memory. S-constant-time programs are still not allowed to branch on secrets, but they can access tables using secret indexes, as long as the tables are stored in stealth memory. As we previously did for constant-time, we will give formal guarantees that the stealth memory mechanism is indeed an effective countermeasure against cache side-channels, and we define a static analysis to check if C programs are S-constant-time. The results in this chapter are the first formal security analysis of stealth memory.

We will start by presenting, in Section 5.1, an extension of the virtualization model with the components needed to represent the stealth memory regions and the platform actions required to handle allocation and access to those regions. Section 5.2 presents a generalization of our isolation result from Chapter 3, that guarantees non-leakage through cache side-channels, when the victim correctly uses the stealth mechanism. In Section 5.3, we extend the static analysis of C programs to check that victim implementations are S-constant-time. Finally, Section 5.4 provides the main non-leakage result showing that program that pass the static analysis are protected against cache side-channels.

5.1 Extensions to the virtualization model

The virtualization model requires modifications analogous to the ones done on a real system, described by the StealthMem authors. We need to define and reserve a cache line set for stealth usage, modify the semantics of the `new` action to add the checks that guarantee that the stealth memory is indeed protected; introduce a new primitive to allocate stealth memory (the `new_sm` action), and modify the behavior of the context switch to restore the stealth page to the cache in the new execution context. As expected, the `read` and `write` actions require no modification. For efficiency, these actions need to execute directly on hardware, without hypervisor intervention: a call to the hypervisor for every memory access would make the mechanism practically unusable. In addition, stealth memory does not require specialized hardware, so it is important to be able to model it without changing the semantics of these two actions.

5.1.1 State extensions

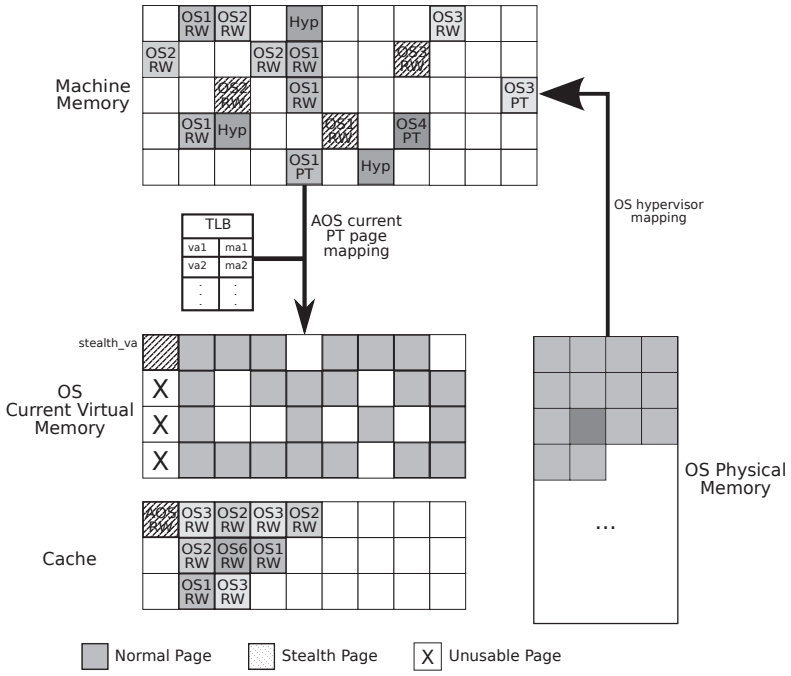

Figure 5.2: Memory model of the platform with stealth memory

We start by presenting extensions on the model states. In Figure 5.2 we show a high level diagram of the new memory model. Note the similarities with Figure 5.1, even though we use only one core, as discussed in Section 2.1.2. One cache index and one particular virtual address (*stealth_va*) in its cache line set is selected for stealth use. All other virtual addresses in that cache line set are reserved and cannot be used either by the guest operating systems or the hypervisor. We allow one stealth page per virtual address space, so each guest operating system can use a set of stealth pages, one for each page table. Each application running on top of the operating systems, however, only has access to one stealth page.

Note that the use of only one stealth page per application is usually not an important restriction; given the intended usage of the stealth memory mechanism, the amount of information that is stored in stealth pages is small. It is usually used to store the look-up tables of cryptographic algorithms, which requires only a small amount of memory, see Chapter 6 for the stealth memory

requirements of some implementations. Furthermore, it is relatively straightforward to extend the definitions to use a set of stealth addresses.

Stealth memory works by disallowing the use of virtual addresses that are on the same cache line set as *stealth_va* (referred to as the *exclusion* property in the StealthMem article [80]), and storing in cache the stealth page of the current operating system on context switches. To guarantee that these restrictions hold in our model, we add four conditions to the valid state invariant:

11. every stealth page is readable/writable and cacheable
12. the current stealth page of the active OS is always cached;
13. if an entry is cached in the stealth cache line set, it must be the stealth page of the active OS.
14. No virtual address in the stealth cache line set, different from *stealth_va*, can be used in the memory space of any operating system.

The proof of these new invariants critically relies on the *inertia* property of caches [80]: upon adding a virtual address to the cache, the evicted virtual address, if any, is in the same cache line set as the added one. Together with the exclusion property this implies that adding a new non-stealth entry to the cache will never evict the stealth page currently cached.

5.1.2 New action semantics

We split the `new` action in two: one for allocating stealth memory (`new_sm`), and one for regular memory allocation of non-stealth pages (`new`). The `new_sm` action allocates a page to the *stealth_va*, and immediately adds it to the cache. The behavior of the non-stealth `new` action is modified to check that the virtual address allocated is not in the stealth cache line set.

We also modify the semantics of the context switches (`switch` and `lswitch` actions), in order to add the stealth page to the cache on context switch (and remove the old one, if necessary).

As mentioned before, the `read` and `write` actions do not need to be modified. Also, the default behavior of the `del` action is correct for both stealth and non-stealth pages, so there is no need to modify it or add an specialized delete action for the stealth page.

The operational semantics of the new actions are presented in the same layout as in section 2.3, with the modifications shown in bold-face.

Action `new` *va pa*

Hypervisor extends the memory of the active OS with $va \mapsto ma$

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \quad os\_accessible(va) \quad oss[aos] = (pa', New\ va\ pa) \\ get\_page\_hyp(s, aos, pa) = (ma, pg) \quad \textbf{non\_stealth\_cache\_line}(\textbf{va}) \\ \neg memory\_alias(mem, va, ma) \\ get\_page\_hyp(s, aos, pa') = (ma', cpt) \quad cpt[va := ma] = cpt' \\ oss[aos := (pa', None)] = oss' \quad mem[ma' := cpt'] = mem' \\ remove\_cache\_va(cache, cpt, va) = cache' \quad tlb[va := \bot] = tlb' \end{array}}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{new}\ va\ pa, act} (oss', act, hyp, mem', cache', tlb')}$$

$$\frac{\begin{array}{c} act = (aos, waiting) \quad os\_accessible(va) \quad oss[aos] = (pa', New\ va\ pa) \\ get\_page\_hyp(s, aos, pa) = (ma, pg) \quad \textbf{non\_stealth\_cache\_line}(\textbf{va}) \\ memory\_alias(mem, va, ma) \\ get\_page\_hyp(s, aos, pa') = (ma', cpt) \quad cpt[va := ma] = cpt' \\ oss[aos := (pa', None)] = oss' \\ pg = (t, o, b) \quad mem[ma := (t, o, false)][ma' := cpt'] = mem' \\ remove\_cache\_va(cache, cpt, va) = cache' \\ remove\_cache\_ma\ cache'\ ma = cache'' \\ tlb[va := \bot] = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{new}\ va\ pa, act} (oss', act, hyp, mem', cache'', tlb')}$$

Precondition

The precondition of this action is the same as before, with the extra condition that the allocated virtual address *va* is not in the stealth cache line set (predicate $non\_stealth\_cache\_line(va)$).

Postcondition

No changes to the postcondition were necessary.

Notes

In order to achieve non-leakage of stealth data, it is necessary that accesses to the *stealth_va* do not replace any entry in the cache. The hypervisor ensures that this is the case by enforcing the *exclusion* property: it only allows operating systems to allocate virtual addresses that are not in the same cache line set as *stealth_va*.

Action `new_sm` *stealth_va pa*

Hypervisor extends the stealth memory of the active OS with $stealth\_va \mapsto ma$

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad os\_accessible(va) \qquad oss[aos] = (pa', New\ stealth\_va\ pa) \\ get\_page\_hyp(s, aos, pa) = (ma, pg) \qquad \mathbf{pg} = (\mathbf{RW}\ \_, \_, \mathbf{true}) \\ \neg memory\_alias(mem, stealth\_va, ma) \\ get\_page\_hyp(s, aos, pa') = (ma', cpt) \qquad \mathbf{cpt}[\mathbf{stealth\_va}] = \bot \\ oss[aos := (pa', None)] = oss' \qquad cpt[stealth\_va := ma] = cpt' \qquad mem[ma' := cpt'] = mem' \\ \mathbf{cache\_add}(\mathbf{cache}, \mathbf{stealth\_va}, \mathbf{ma}, \mathbf{pg}) = (\mathbf{cache'}, \_) \\ \mathbf{tlb}[\mathbf{stealth\_va} := \mathbf{ma}] = \mathbf{tlb'} \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{new\_sm}\ stealth\_va\ \ pa, act} (oss', act, hyp, mem', cache', tlb')}$$

Precondition

The action `new_sm` *stealth_va pa* has the same precondition as the `new` action for the not aliased case; with the extra requirement for the new page to be readable/writable and cacheable.

Postcondition

The only additions to the postcondition is that the new stealth page is immediately stored in the *cache*, and the mapping in the TLB is updated.

Notes

Allocating an aliased stealth page is not allowed, since the stealth page *pg* in *ma* must always remain cached (aliased pages are not cacheable in our model). This is why we only define one `new_sm` rule for the case when no alias is generated.

Action `switch` o

Hypervisor sets o to be the active OS

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad oss[o] = (pa, None) \qquad get\_page\_hyp(s, o, pa) = (\_, cpt) \\ (o, waiting) = act' \qquad \mathbf{stealth\_save}(\mathbf{mem}, \mathbf{cache}) = \mathbf{mem'} \\ \mathbf{stealth\_add}(\mathbf{stealth\_drop}(\mathbf{cache}), \mathbf{mem}, \mathbf{o}, \mathbf{cpt}) = \mathbf{cache'} \\ tlb\_flush(tlb) = tlb' \end{array}}{(oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{switch}\ o,o} (oss, act', hyp, mem', cache', tlb')}$$

Precondition

The precondition of this action is the same as before.

Postcondition

In the resulting state, the stealth page of the new active OS, if any, is cached immediately, after saving the old stealth cache page into memory.

Notes

The semantics of this action reflects a distinguishing characteristic of VIPT caches, i.e. the cache is not flushed on context switches. As a consequence, the execution of one process of an OS may result in the eviction from the cache of the entries belonging to another concurrent OS; as we have seen, this behavior can be exploited by an attacker to gain information on the execution of a victim OS. The fact that the stealth page of the active OS is always cached is what guarantees that these attacks cannot be performed on stealth pages.

Action `lswitch` *pa*

Hypervisor changes the current memory mapping of the active OS to be *pa*

Rule

$$\frac{\begin{array}{c} act = (aos, waiting) \qquad oss[aos] = (pa', LSwitch\ pa) \\ get\_page\_hyp(s, aos, pa) = (ma, pg) \\ pg = (PT\ \_, OS\ aos, \_) \qquad get\_page\_hyp(s, aos, pa') = (\_, cpt) \\ oss[aos := (pa, None)] = oss' \qquad \mathbf{stealth\_save(mem, cache)} = \mathbf{mem'} \\ \mathbf{stealth\_add(stealth\_drop(cache), mem, aos, cpt)} = \mathbf{cache'} \\ tlb\_flush(tlb) = tlb' \end{array}}{s = (oss, act, hyp, mem, cache, tlb) \xrightarrow{\texttt{lswitch}\ pa,act} (oss', act, hyp, mem', cache', tlb')}$$

Precondition

The precondition of this action remains unchanged.

Postcondition

As in the `switch` action, the cached stealth page of the previous process is saved in memory, and the stealth page of the new process (mapped by the new current page table) is immediately stored in the cache.

5.2 Isolation

Stealth memory must allow operating systems to access memory regions without leaking its usage patterns. This means that accesses to the *stealth_va* should not be observable by the attacker. Isolation in this setting is therefore stated as a non-interference property, where not only the attacker must be unable to distinguish between states in two executions, but also the attacker must not be able to observe an access of the victim to the stealth region. In other words, the execution of an access to the stealth region should not influence the attacker behavior in any way. As mentioned in Section 3.2, this is a generalization of the *non-leakage* result of theorem 3.6.1, resulting in a *non-influence* property.

We classify the platform actions into *stealth actions* and *non-stealth actions*. Stealth actions are the ones that use or change the *stealth_va* virtual address: `read` and `write`, when the accessed address is stealth; `new_sm`; `del`, when the deleted address is stealth; and local context switches, which change the current page table of the active OS, and therefore the cached stealth page. We define the effect of a stealth action to be the empty (the same as the `silent` action), so that two arbitrary stealth actions always have the same effect.

In this new setting, the attacker should be completely unaware of the stealth memory regions of the victim. We will therefore update our equivalence relation for states, to require that only the non-stealth part of the memory of the victim be equivalent. This new definition allows the victim to have arbitrary stealth regions in both states; or even to have a stealth page in one state but not the other. Figures 3.1 and 3.2 provide illustration for these definitions, and are mainly unchanged in the new setting. The new formal definition of each equivalence condition are as follows:

Definition 5.2.1 (Equivalence of *non-stealth* hypervisor mappings)**.** Two states s and s' have equivalent non-stealth hypervisor mappings for the attacker ($s \sim^{hyp} s'$) if for every physical address *pa*, readable/writable page *pg* and machine address *ma*:

- if $get\_page\_hyp(s, o_a, pa) = (ma, pg)$, there exists ma' such that $get\_page\_hyp(s', o_a, pa) = (ma', pg)$;
- if $get\_page\_hyp(s, o_v, pa) = (ma, pg)$, **and no page table maps** $stealth\_va$ **to** ma, then there exists ma' such that $get\_page\_hyp(s', o_v, pa) = (ma', pg')$, where pg and pg' are equal except in their contents;

and reciprocally for s'.

Definition 5.2.2 (Equivalence of *non-stealth* cache and memory mappings)**.** Two states s and s' have equivalent non-stealth cache and memory mappings for the attacker ($s \sim^{mem} s'$) if for every physical address pt_pa, page table pt_pg and machine address pt_ma, such that $get\_page\_hyp(s, o, pt\_pa) = (pt\_ma, pt\_pg)$, there exist pt_ma' and pt_pg' such that $get\_page\_hyp(s', o, pt\_pa) = (pt\_ma', pt\_pg')$ and for all virtual address va and machine address ma such that $pt\_pg[va] = ma$ and $get\_page\_va\_ma(s, va, ma)$ is readable/writable, then there exists ma' such that $pt\_pg'[va] = ma'$. Furthermore, the same physical address pa that maps to ma in s, maps to ma' in s'; and moreover:

- if $o = o_a$ then $get\_page\_va\_ma(s, va, ma) = get\_page\_va\_ma(s, va, ma')$;
- if $o = o_v$, and va **is not** $stealth\_va$ then $get\_page\_va\_ma(s, ma)$ and $get\_page\_va\_ma(s', ma')$ are equal except in their contents;

and reciprocally for s'.

Definition 5.2.3 (Equivalence of *non-stealth* cache histories)**.** Two cache histories $history$ and $history'$ are equivalent for the attacker if for all **non-stealth** cache index i, $history[i] = history'[i]$.

The statement of the two step-consistent unwinding lemmas (3.5.1 and 3.5.2) remain the same, but they now use the new equivalence definition and must therefore be proved again. We show the statements again here:

Lemma 5.2.4 (o_a step-consistent unwinding lemma)**.** Let s_1 and s_1' be states where o_a is active, and such that $s_1 \xrightarrow{a,o_a} s_2$, $s_1' \xrightarrow{a,o_a} s_2'$ and $s_1 \sim s_1'$. Then, $s_2 \sim s_2'$.

Lemma 5.2.5 (o_v step-consistent unwinding lemma)**.** Let s_1 and s_1' be states where o_v is active, and such that $s_1 \xrightarrow{a,o_v} s_2$, $s_1' \xrightarrow{a',o_v} s_2'$, $\mathsf{eff}(a) = \mathsf{eff}(a')$ and $s_1 \sim s_1'$. Then, $s_2 \sim s_2'$.

Proof. Again, to prove these lemmas each action must be analyzed checking the equivalence conditions on the resulting states. For the unchanged actions, the previous proofs can be reused.

The actions with new semantics are also easily proved using the previous results. The `new` action is the same as before, with an added precondition, so the result still holds. The `new_sm` is like executing a normal `new`, and then performing a `read` of the stealth page. Composing both executions, the resulting states are proved to be equivalent. Similarly, the `switch` and `lswitch` can be viewed as an execution of the old context switch, with a `read` of the stealth page afterwards.

In this manner, these two lemmas can be easily proved by reducing them to the non-stealth case. □

In addition, as is required of a non-influence result, we need to prove one more unwinding lemma that shows that the execution of a stealth action is unobservable by the attacker:

Lemma 5.2.6 (Locally preserves unwinding lemma)**.** Let s and s' be states such that $s \xrightarrow{a,o_v} s'$ and $\mathsf{eff}(a) = \emptyset$. Then, $s \sim s'$.

Proof. As in the previous two unwinding lemmas, we need to analyze each action, checking all equivalence conditions for the states.

For a `read` or `write` action to the *stealth_va*, the stealth page is guaranteed to be always cached (by condition 12 of valid state, and because the victim is active and running). Therefore, upon execution of one of these actions, the state remains the same, except possibly for the value written in the stealth cache entry in the case of a `write`. The states before and after the execution of any of these actions can therefore be shown to be indistinguishable for the attacker.

In the case of the `new_sm` and `del` actions, the non-stealth part of the victim memory is shown to be unchanged, except for the new or removed mapping from *stealth_va* in the current page table of the victim. This is taken into account in the equivalence definition, and the states are therefore indistinguishable.

For the `lswitch`, the non-stealth part of the victim memory is shown to remain unchanged. The only possible state change is the caching of a new stealth page, and the writing-back of the old stealth cache entry; and both changes modify the stealth (unobservable) part of the victim memory. □

The condition for the victim that required that it executes actions with the same effects on both traces ($v\_str \approx v\_str'$) remains the same; but now using the new definition of the effects of the actions:

$$\frac{\mathsf{eff}(a) = \mathsf{eff}(a') \quad v\_str \approx v\_str'}{(a :: v\_str) \approx (a' :: v\_str')}$$

This rule requires the victim to execute two non-stealth actions with the same effect (i.e. two writes to the same virtual address but with arbitrary value, or the exact same action), or two arbitrary stealth actions or silent (it can execute a stealth action in one trace, and silent in the other).

Analogously, the lifting of the equivalence relation for traces is the same as before, using the new definitions of action effects and state equivalence:

$$\frac{s \sim s' \quad \Theta \sim \Theta'}{(s \xleftarrow{a,o_a} \Theta) \sim (s' \xleftarrow{a,o_a} \Theta')}$$

$$\frac{\mathsf{eff}(a) = \mathsf{eff}(a') \quad s \sim s' \quad \Theta \sim \Theta'}{(s \xleftarrow{a,o_v} \Theta) \sim (s' \xleftarrow{a',o_v} \Theta')}$$

The rules allow executing the same attacker action, or two victim actions with the same effect (either both non-stealth with the same effect, or two arbitrary stealth actions), from two equivalent states. The statement of the isolation theorem remains the same, but using these new definitions:

Theorem 5.2.7 (S-constant-time isolation)**.** Let s_0 and s_0' be two states such that $s_0 \sim s_0'$; v_str and v_str' two sequence of victim actions such that $v\_str \approx v\_str'$; and Θ and Θ' be traces such that $interleave(s_0, v\_str, \Theta)$ and $interleave(s_0', v\_str', \Theta')$. Then $\Theta \sim \Theta'$.

Proof. As was the case for theorem 3.6.1, this proof is a direct consequence of the unwinding lemmas and a co-inductive argument.

Indeed, when the attacker selects his own action to execute, by the co-inductive hypothesis, it does so from equivalent states and equivalent partial traces. Therefore, the selected attacker action must be the same in both traces. Lemma 5.2.4 allows us to prove the equivalence of the first state of the tail of the traces.

When the attacker selects victim actions to execute, since they have the same effects, they are either both non-stealth with the same effect, allowing us to use Lemma 5.2.5; or two arbitrary

stealth actions, allowing us to use Lemma 5.2.6 twice, once for each trace, and proving the equivalence by the transitivity of the state equivalence. □

Note that even though the attacker is not able to observe the stealth memory of the victim, we cannot allow the victim to execute completely arbitrary stealth actions, and both traces must still be in lockstep. In other words, if the victim executes a stealth action in one trace, it must do so in the other trace, or execute `silent` in its place. This means that, even using stealth memory, we still cannot allow high branches in victim code.

To illustrate the problem, suppose the victim performs a read inside an if with a High condition, and then a read to some variable:

```
...
if <high_condition> {
  read Xs
}
read Y
...
```

In this case, the variable `Xs` must be stealth. Otherwise, the victim would execute the non-stealth `read` to `Xs` in one trace (when the condition is true) and not in the other (if the condition is false), and this is not safe. Variable `Y` need not be stealth, as it is not inside a high region. This program is still not secure. Indeed, the trace of victim actions would be:

$$\begin{array}{lll} v\_str & = (\texttt{read Xs}) :: & (\texttt{read Y}) :: ... \\ v\_str' & = & (\texttt{read Y}) :: ... \end{array}$$

When the scheduler interleaves these actions, it must select one action from both traces. The resulting traces could then be:

$$\begin{array}{lll} \Theta & = s_1 \xrightarrow{\texttt{read Xs}, o_v} & s_2 \xrightarrow{a, o_a} s_3 \xrightarrow{\texttt{read Y}, o_v} s_4 ... \\ \Theta' & = s'_1 \xrightarrow{\texttt{read Y}, o_v} & s'_2 \xrightarrow{a, o_a} s'_3 ... \end{array}$$

Note that the attacker executes action a from two *non-equivalent* states s_2 and s'_2 which breaks isolation. The problem is not in the visibility of the stealth action `read Xs` (which we proved to be non-interferent in Lemma 5.2.6), but in the visibility of the non-stealth action `read Y`.

This happens even if the attacker has no control over the scheduler (for example, a round robin scheduler that executes exactly one action of each operating system would generate these traces). The root of the problem lies in the code having a non-balanced high branch, so the scheduler can let the attacker execute when the victim is still inside the High region in one trace and in a Low region in the other.

Padding the shorter branch (in this case the empty `else` branch) with silent actions would make the program secure again in our model. However, this technique is not very realistic as usual schedulers work with time slices instead of discreet actions. In an actual execution, then, timing differences between the executions will be the main source of leakage.

Coppens et al [48] argue that it would be extremely complex (or even impossible) to guarantee that two executions take the exact same time in a modern processor. In that work they propose the use of if-conversion in order to eliminate high branches when the program is compiled.

Since high branching is generally considered harmful in cryptography, we decided to disallow them entirely for S-constant-time programs. This restriction is not a problem for the algorithms we analyze in Chapter 6, but trying to remove it is an interesting line of future work.

5.3 A type system for stealth memory

We have shown that the attacker has no visibility over the stealth memory region of the victim. We can therefore relax our type system to allow secret-dependent memory accesses on stealth addresses. The modified typing rules now involve a set X_s of addresses that must be mapped to stealth memory. The main typing rules are now given in Figure 5.3. Note that there is no requirement that stealth addresses are high; in practice, stealth addresses often store public tables.

$$\frac{p(n) = \mathsf{load}_\varsigma(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \qquad \tau(\vec{r}) = \mathsf{High} \implies \mathcal{S} \in X_s}{X_s, X_h \vdash n : \tau \Rightarrow \tau[r \mapsto \tau(\vec{r}) \sqcup X_h(\mathcal{S})]}$$

$$\frac{p(n) = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \qquad PointsTo(n, addr, \vec{r}) = \mathsf{Symb}(\mathcal{S}) \qquad \tau(\vec{r}) = \mathsf{High} \implies \mathcal{S} \in X_s \qquad \tau(\vec{r}) \sqcup \tau(r) \sqsubseteq X_h(\mathcal{S})}{X_s, X_h \vdash n : \tau \Rightarrow \tau}$$

Figure 5.3: Information flow rules for S-constant-time

Definition 5.3.1 (S-constant-time)**.** A program p is S-constant-time with respect to a set of variables X_h^0 and a set of stealth addresses X_s, written $X_s, X_h^0 \vdash p$, if there exists (X_h, T) such that for every $\mathcal{S} \in X_h^0$, $X_h(\mathcal{S}) = \mathsf{High}$ and for all nodes n and all its successors n', there exists τ such that

$$X_s, X_h \vdash n : T(n) \Rightarrow \tau \quad \wedge \quad \tau \sqsubseteq T(n')$$

where $\sqsubseteq$ is the natural lifting of $\sqsubseteq$ from $\mathbb{L}$ to to types.

As before, the sets X_h and T are inferred using Kildall's algorithm.

Our soundness result is extended to the setting of stealth memory simply by considering a modified labelled operational semantics (see Figure 5.4) where accessing variables in X_s has no visible effect; the equivalence of language states, that required states to have the same values for the Low variables, remains unmodified.

$$\frac{p[n] = \mathsf{load}_\varsigma(addr, \vec{r}, r, n') \qquad [\![addr]\!](\rho, \vec{r}) = v_{\text{addr}} \qquad v_{\text{addr}} \notin X_s \qquad \mu[v_{\text{addr}}]_\varsigma = v}{(n, \rho, \mu) \xrightarrow{\texttt{read}\ v_{\text{addr}}} (n', \rho[r \mapsto v], \mu)}$$

$$\frac{p[n] = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \qquad [\![addr]\!](\rho, \vec{r}) = v_{\text{addr}} \qquad v_{\text{addr}} \notin X_s \qquad store(\mu, \varsigma, v_{\text{addr}}, \rho(r)) = \mu'}{(n, \rho, \mu) \xrightarrow{\texttt{write}\ v_{\text{addr}}} (n', \rho, \mu')}$$

$$\frac{p[n] = \mathsf{load}_\varsigma(addr, \vec{r}, r, n') \qquad [\![addr]\!](\rho, \vec{r}) = v_{\text{addr}} \qquad v_{\text{addr}} \in X_s \qquad \mu[v_{\text{addr}}]_\varsigma = v}{(n, \rho, \mu) \xhookrightarrow{\emptyset} (n', \rho[r \mapsto v], \mu)}$$

$$\frac{p[n] = \mathsf{store}_\varsigma(addr, \vec{r}, r, n') \qquad [\![addr]\!](\rho, \vec{r}) = v_{\text{addr}} \qquad v_{\text{addr}} \in X_s \qquad store(\mu, \varsigma, v_{\text{addr}}, \rho(r)) = \mu'}{(n, \rho, \mu) \xhookrightarrow{\emptyset} (n', \rho, \mu')}$$

Figure 5.4: Modified IR semantics (excerpts)

The soundness statement is the same as before, except that now all actions involving a stealth variable have the $\emptyset$ effect.

Theorem 5.3.2 (Soundness of S-constant-time verification)**.** If $X_s, X_h^0 \vdash p$ then For every two executions of p:

$$t_1 \xrightarrow{e_1} t_2 \xrightarrow{e_2} t_3 \xrightarrow{e_3} \dots \qquad\qquad t'_1 \xrightarrow{e'_1} t'_2 \xrightarrow{e'_2} t'_3 \xrightarrow{e'_3} \dots$$

if $t_1 \sim_{X_h} t'_1$, then for all i, $t_i \sim_{X_h} t'_i$ and $e_i = e'_i$.

Proof. If the program type-checks, it can be shown, by the same argument as before, that after the execution from two indistinguishable states of two instructions with the same effect (two non-stealth reads or writes to the same addresses, or two instructions with no effect) the resulting states will also be indistinguishable.

In order to show that instruction effects are the same in both traces, we note that High branches are still forbidden (and therefore the program has to take the same path in both executions), and that expressions used as addresses in all instructions are either Low (and therefore equal), or High (and therefore, the base address must be stealth). In the second case, the resulting effect is $\emptyset$ in both executions, so they are also equal. □

5.4 System-level security for stealth memory

As before, we can link the type system and the semantics of the program in the platform model. The mapping from MachIR states to platform states must now map elements of X_s to stealth addresses. Under this condition, we can prove that if a program p is S-constant-time ($X_s, X_h^0 \vdash p$) with respect to the set of secrets X_h^0, then an attacker $\mathcal{A}$ cannot distinguish between two different executions of this program from two initial states that are indistinguishable and have the same values for all Low variables. Formally:

Theorem 5.4.1 (System-level security)**.** Given a program p, an attacker $\mathcal{A}$, and initial states s_0 and s'_0; such that $s_0 \sim s'_0$ and $s_0 \sim_{X_h^0} s'_0$. If $X_s, X_h^0 \vdash p$, then $(\mathcal{A} \parallel p)[s_0] \sim (\mathcal{A} \parallel p)[s'_0]$.

Proof. Again, we can combine the results from theorems 5.3.2 and 5.2.7. Since we are in the hypothesis of the first, we can prove that the effects of the victim actions are the same in both executions and therefore, by Theorem 5.2.7, the attacker cannot distinguish between the two executions. □

As mentioned before, the sets X_h and T are inferred using Kildall's algorithm. The set X_s can also be computed modifying the algorithm to track array accesses with high indices. We do not do this at the moment, but we have done it for an earlier version, and we are confident it would be easy to add the functionality.

In addition, the compiler can be modified to perform stealth allocation automatically, producing an S-constant-time object program with a formal guarantee that it does not leak secret information. However, as this last modification would be dependent on the specific architecture and the stealth memory implementation used, it was not included and is left as future work.

To make the presentation clearer, in this thesis we chose to separate the constant-time analysis from the stealth memory part. The actual CompCert implementation, however, was done with stealth memory primitives from the beginning, and not as an extension. Since one can think of constant-time programs as a particular case of S-constant-time with no stealth memory regions available, we modeled the stealth static analysis first, and then handled constant-time by disabling the use of stealth memory. In other words, if a program type-checks without using any stealth addresses, it is constant-time and proven to be non-leaking.

Chapter 6

Applications to cryptographic algorithms

In this chapter we present the results of applying our static analyses to a representative set of cryptographic implementations, including some that are vulnerable to cache-based attacks on common platforms, and constant-time ones that were specifically designed to avoid such attacks. In all cases, we considered publicly available implementations of the algorithms, compiled them using CompCert, and run our certified type system on the MachIR programs output by the compiler.

We start by giving an overview of the verification process and results in Section 6.1. Section 6.2 contains a brief description and specific results of each verified algorithm. We use the AES algorithm in Section 6.3 as a study case to show the verification process in detail.

6.1 Overview

As described in Chapter 4, our static analysis is implemented on top of the CompCert compiler. In order to type check a C source code file, the user should specify the names of the variables that must be protected (the initial set of High variables) and the set of variables that should be allocated in stealth memory. This set should be empty when type checking constant time programs. If it compiles successfully, the program is guaranteed to be safe against the type of cache side-channel attacks modeled in this work.

Figure 6.1 summarizes the list of examples analyzed, and provides in each case the number of variables allocated as stealth, and the amount of stealth memory that is required to execute the program securely.

As shown in the figure, a wide variety of cryptographic implementations could be analyzed. As expected, some examples required minor rewriting of the program in order to fit with our alias type system (specifically, we had to rewrite programs so that arrays become global variables).

After this rewriting, a test case of the algorithm was selected using some test vectors provided by the libraries used; and the set of secrets (in general the cryptographic key) was identified. The execution of CompCert was done using flags added for our analysis. These include the `-secrets` flag to mark given global variables as secrets, an optional flag `-stealths` to consider the given global variables as stealths, and `-dmachir` to dump the generated MachIR code in a file. The execution of the compiler can fail because of syntax errors or limitations of the alias analysis, or because the program is not (S-)constant-time. If it compiles correctly, the executable is guaranteed to be immune to the cache based attacks we modeled in this work.

Example	LoC	# addresses	Size (KB)
DES	836	10	2
BlowFish	279	1	4
AES	744	5	4
RC4	164	1	0.25
Snow	757	6	6
Salsa20	1077	-	-
TEA	70	-	-
SHA256	419	-	-

Table 6.1: Selected experimental results

6.2 Evaluated algorithms

6.2.1 AES

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm that was selected by NIST in 2001 to replace DES [2]. AES is very widely used and it is expected to remain the prevailing blockcipher for the next 20 years. Although NIST claimed the selected algorithm resilient against side-channels, AES is a prominent example of an algorithm in which the sequence of memory accesses depend on the cryptographic key.

As a testcase for our approach, we have applied our stealth verification to the PolarSSL (now mbed TLS [16]) implementation of AES. Our type system is able to prove that 4kB of stealth memory is sufficient to execute AES securely. In Section 6.3 we explain this algorithm and show the result of our static analysis in detail.

6.2.2 DES and BlowFish

Data Encryption Standard (DES) [1] and BlowFish [112] are symmetric encryption algorithms that were widely used until the advent of AES. They are designed under the same principles as AES, and their implementation critically relies on S-boxes. Cache-based attacks against DES and BlowFish are reported by Tsunoo et al [121] and Kelsey et al [78] respectively.

We have applied our stealth verification to the PolarSSL implementation of DES, and the C implementation of BlowFish by Bruce Schneier [113]; again, our tool proves that only a small amount of stealth memory (resp. 2kB and 4kB) is required for the programs to execute securely.

6.2.3 SNOW

Snow [4] is a stream cipher used in standards such as the 3GPP encryption algorithms. Its implementation relies on table look-ups for clocking its linear feedback shift register (LFSR). Cache-based attacks against SNOW—and similar LFSR-based ciphers—are reported by Leander, Zenner, and Hawkes [91]. We have applied our stealth verification on an ECRYPT implementation of SNOW; our tool proves that SNOW can be executed securely with 6kB of stealth memory.

6.2.4 RC4

RC4 is a stream cipher introduced by Rivest in 1987 and used in cryptographic standards such as SSL and WPA. It is based on a pseudo-random generator that performs table look-ups. Chardin, Fouque and Leresteux [42] present a cache-based attack against RC4. Analyzing the PolarSSL

implementation of RC4 with our stealth verification proves that the program can execute securely with only 0.25kB of stealth memory.

6.2.5 TEA, Salsa20, SHA256

We have applied our constant-time type system to some cryptographic algorithms that carefully avoid performing table look-ups with indices dependent on secrets: Tiny Encryption Algorithm [127], a block cipher designed by Needham and Wheeler; Salsa20 [30], a stream cipher designed by Bernstein, and SHA256 [3]. For the latter, we consider the input text to be secret, with the intention to demonstrate that SHA256 is suitable to be used in password hashing. In all cases, our type system establishes that the programs are indeed constant-time, and therefore secure without using stealth memory.

6.3 Detailed example: AES

For concreteness, we consider AES-128, i.e. the variant of AES with keys of 128 bits; other variants use other key lengths but operate under similar principles. AES-128 encrypts and decrypts messages by iteratively repeating 10 rounds of computation. Each AES round processes 128 bits blocks, which can conveniently be viewed as a 4×4 bytes matrices, using a round key, which can also be viewed as a 4×4 bytes matrix and is obtained from the AES key using a key expansion algorithm.

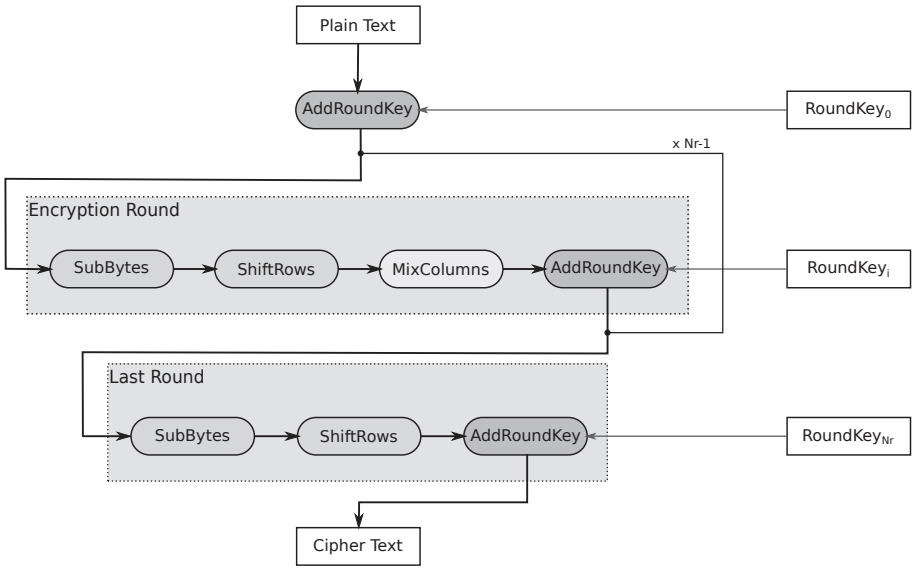

Figure 6.1: AES encryption diagram

Figure 6.1 illustrates the operation of the encryption algorithm. At each round, the algorithm turns a 4×4 matrix into another 4×4 matrix, by performing the following steps: performing pointwise replacement according to some fixed substitution (`SubBytes`), shifting the rows of the input matrix (`ShiftRows`), multiplying the output with a fixed 4×4 matrix (`MixColumns`) and computing the pointwise *exclusive or* of the resulting matrix and of the round key (`AddRoundKey`). The last round is special in that the `MixColumns` operation is not performed.

Most applications of AES require encryption and decryption to be efficient. Software implementations of AES achieve a high degree of efficiency by storing in the cache look-up tables, as recommended by the AES specification. For instance, many implementations rely on look-up

tables, known as S-boxes, to perform substitutions efficiently. More efficient implementations, such as the PolarSSL implementation, store intermediate computations in eight 16×16 tables of 32-bits values, and as a result, can bypass expensive arithmetic operations in the field GF(2^8). The standard security notion for key-alternating blockciphers such as AES is indinstinguishability from a random permutation. Informally, it states that an adversary has small probability to distinguish between a stream of bitstrings output by the encryption scheme or by a random permutation. This assumption is widely used in the provable security analysis of schemes that build on AES, and best-known cryptanalyses on AES only marginally improve on brute force attacks (the estimated complexity of the biclique attack from [36] is $2^{126.2}$) or target reduced rounds AES (e.g. [59] exhibit a chosen-key distinguisher for 9-rounds AES-128).

However, many AES implementations are vulnerable to cache-based timing attacks, and fail to comply with even the weakest security guarantees. Already in 1999, Koeune and Quisquater [86] report timing attacks against careless implementations of AES in which conditional statements depend on secret data—a first recipe for security disaster. In a nutshell, these implementations exploit an algebraic equality between two tables S and S':

$$S'[b] = \begin{cases} 2S[b] & \text{if } S[b] < 128 \\ 2S[b] \oplus 283 & \text{if } S[b] \geq 128 \end{cases}$$

to compute S' from S using a conditional statement that tests $S[b] < 128$, where b is secret. As noted by Koeune and Quisquater [86], the attack is easily circumvented. In particular, high-speed implementations of AES pre-compute the values of the tables, and hence the attack is moot.

However, these implementations still perform table accesses that depend on secret data—a second recipe for a security disaster. In 2005, Bernstein [31] reports on a simple timing attack which allows to recover AES keys by exploiting the correlation between execution time and cache behavior during computation. Because memory accesses depend on the secret key, attackers can recover the secret key by measuring the execution time for sufficiently many AES computations. Shortly afterwards, Tromer, Osvik, and Shamir [120] concurrently report on several attacks against AES, including an effective attack that does not require knowledge of the plaintexts or the ciphertexts. Further improvements of the attacks have been reported, notably by Bonneau and Mironov [37], by Aciiçmez, Schindler and Koç [7], and by Canteaut, Lauradoux and Seznec [41]. Bangerter, Gullasch and Krenn [68] report on an attack in which key recovery is performed in almost real-time, and Ristenpart et al [109] show that cache-based attacks can be realized in virtualized cloud platforms.

6.3.1 Example C implementation

In this section we present part of the C source code of the AES encryption function from the PolarSSL library.

The code uses a known optimization on 32 bit processors [50] to combine the `SubBytes`, `ShiftRows` and `MixColumns` steps of each round into appropriate table look-ups. Four 256-entry tables, here called `FT0`, `FT1`, `FT2` and `FT3`, are needed for this optimization. Each AES round is implemented in the `AES_FROUND` macro, which also performs the `XOR` with the round key to implement the `AddRoundKey` step. In lines 20 to 23, the first `AddRoundKey` of the algorithm is performed. Then, the `for` in line 25 executes $Nr - 2$ rounds. An additional round is done on line 31. Finally, the special last round is done from line 33 onward, in which the S-box `FSb` is used instead of the `FTi` tables, since there is no `MixColumns` step in this round.

```
1  #define AES_FROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3)
2  {
3      X0 = *RK++ ^ FT0[ ( Y0       ) & 0xFF ] ^
4                   FT1[ ( Y1 >>  8 ) & 0xFF ] ^
5                   FT2[ ( Y2 >> 16 ) & 0xFF ] ^
6                   FT3[ ( Y3 >> 24 ) & 0xFF ];
7      ...
8  }
9
10 int aes_crypt_ecb( aes_context_nr *ctx_nr, aes_context_rk *ctx_rk,
11                    aes_context_buf *ctx_buf,
12                    const unsigned char input[16],
13                    unsigned char output[16] )
14 {
15   int i;
16   uint32_t *RK, X0, X1, X2, X3, Y0, Y1, Y2, Y3;
17
18   RK = ctx_rk->rk;
19
20   GET_UINT32_LE( X0, input,  0 ); X0 ^= *RK++;
21   GET_UINT32_LE( X1, input,  4 ); X1 ^= *RK++;
22   GET_UINT32_LE( X2, input,  8 ); X2 ^= *RK++;
23   GET_UINT32_LE( X3, input, 12 ); X3 ^= *RK++;
24
25   for( i = (ctx_nr->nr >> 1) - 1; i > 0; i-- )
26   {
27     AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
28     AES_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 );
29   }
30
31   AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
32
33   X0 = *RK++ ^ \
34           ( (uint32_t) FSb[ ( Y0       ) & 0xFF ]       ) ^
35           ( (uint32_t) FSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
36           ( (uint32_t) FSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
37           ( (uint32_t) FSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );
38 ...
39   PUT_UINT32_LE( X0, output,  0 );
40   PUT_UINT32_LE( X1, output,  4 );
41   PUT_UINT32_LE( X2, output,  8 );
42   PUT_UINT32_LE( X3, output, 12 );
43   return( 0 );
44 }
```

Listing 6.1: `aes_crypt_ecb` C source code

Even though the actual static analysis is done on the MachIR compiled code, we will illustrate the reasoning behind our analysis on the C source code to compare it to the actual analysis described in the next section. We will consider the round key (`ctx_rk`) as secret. This variable taints `RK` as High in the assignment of line 18, and `Xi` in lines 20 to 23. In line 27 a value that depends on `Xi` is used to index the `FTi` tables. These tables must therefore be allocated in stealth memory. Furthermore, the variables `Yi` are also tainted at this point. In lines 34 to 37 we have accesses to the `FSb` table that uses `Yi` as indices. Since the `Yi` are High, this table must also be allocated in stealth memory for the algorithm to be considered S-constant-time.

6.3.2 MachIR code

Next, we show a fragment of the compilation of this function to MachIR. The MachIR pretty printer shows the security level of expressions at each program point and the result of the alias analysis when performing a memory access.

```
652  AX{L} = (ctx_rk + 0){L} goto 651
...
624  DX{H} = int32[(AX + 28){L}]{ctx_rk} goto 623
623  SI{H} = DX{H} goto 622
622  SI{H} = (SI >>u 8){H} goto 621
621  SI{H} = (SI & 255){H} goto 620
620  SI{H} = int8u[(FSb + 0 + SI){H}]{FSb} goto 619
```

Listing 6.2: `aes_crypt_ecb` MachIR compilation

The address of `ctx_rk` is first stored in the `AX` register. This address is Low, even if the content of the `ctx_rk` variable is High. Then, in line 624 the address is accessed, and its contents loaded in the `DX` register (note how the points-to analysis infers the variable pointed to by register `AX`). Since the contents of `ctx_rk` are High, the register is also tainted with a High security level. The register `SI` is used to compute the index to access the array `FSb`, using the value of `DX`. Finally, in line 620, `FSb` is accessed using a high index, so it is marked as stealth.

After type checking the rest of the code, the variables `FSb`, `FT0`, `FT1`, `FT2` and `FT3` are required to be stealth. These variables correspond to S-boxes and look-up tables used by the encryption implementation. The total amount of memory needed to store these variables is 4Kb.

Chapter 7

Related Work

In this chapter we discuss some work related to the topics of this thesis, such as OS modeling and verification, side-channels attacks in cryptography and their countermeasures, information flow and language based protection mechanisms and program verification of cryptographic implementations.

7.1 OS verification

OS verification is an active field of research [115]. An important breakthrough was the machine-checked refinement proof of an implementation of the seL4 microkernel, a general purpose operating system of the L4 family [83]. The main thrust of the formal verification is to show that an implementation of the microkernel correctly refines an abstract specification. Subsequent machine-checked developments prove that seL4 enforces integrity, authority confinement [114] and intransitive non-interference [102]. The formalization includes some confidentiality results in the presence of side-channels, but it does not model cache nor timing attacks.

Another very prominent initiative is the Microsoft Hyper-V verification project [47, 92], which has made a number of impressive achievements towards the functional verification of the implementation of the Hyper-V hypervisor, a large software component that combines C and assembly code (about 100 kLOC of C and 5kLOC of assembly). The overarching objective of the formal verification is to establish that a guest operating system cannot observe any difference between executing through the hypervisor or directly on the hardware.

Dam et al [51] formally verify information flow security for a simple separation kernel for ARMv7. The verification is based on an extant model of ARM in HOL, and relates an ideal model in which the security requirements hold by construction with a real model that faithfully respects the system behavior. Extending the approach to handle the cache is left for further work.

IronClad [69] is another project that aims at providing strong security for a complete software stack, including applications, operating system, drivers, and cryptographic libraries. They prove not only functional correctness, but non-interference results as well. They do not, however, model the cache or side-channels that arise from its use.

A significant number of OS verification projects have their primary focus on security. Our work is most closely related to von Oheimb et al's proof of non-leakage for the SLE88 smart card [103, 104], and to isolation proofs for separation kernels [70, 67, 97]. Other related work includes Sewell et al [114]'s proof of integrity for seL4. Recently, Baumann et al [29] establish OS isolation for an implementation of a separation kernel; however, the isolation property is derived by a pen-and-paper proof from the formal verification in VCC.

An alternative to prove the safety and security of legacy operating systems is to design provably safe and secure operating systems. Singularity [72], HiStar [130] and Flume [90] are prime examples of operating systems that were designed with provable safety and security in mind. Krohn and Tromer [89] provide a CSP model of the Flume system [90] and show non-interference of the idealized model; they also discuss covert channels, but do not establish non-interference in presence of leakage. As for legacy systems, machine-checked verification provides stronger confidence in the guarantees achieved by a particular architecture. Tiwari et al [119] and Yang and Hawblitzel [128] use static analysis and type systems to prove security and safety properties of their implementations.

7.2 Memory and OS models

Most works on OS verification (including this thesis) consider an idealized memory model that makes simplifying assumptions, or abstracts away some component. For instance, our work provides a simplified treatment of page tables and elides the possibility of nesting. Such multi-level page tables are modeled in more detail e.g. in the L4.verified project, see [87]; interestingly, Franklin et al [61, 60] provide small model theorems that allow reducing the verification of a certain class of properties from models that consider nested page tables to models with page tables of depth 1. Shadow page table algorithms are also considered in [11], where the authors provide a formal model of the TLB. In addition, there is a number of relevant formalization efforts that have considered the TLB and the cache, but that have not yet integrated these components in their formal models. For instance, Tews et al [118] develop a stack of memory models, and prove formally that the models satisfy a form of integrity property, called the plain memory property. They discuss a TLB memory model, but leave it for future work. Likewise, Kolanski [87] describes as future work a possible approach to account for the TLB and cache in the seL4 model.

Other OS components and mechanisms that have been studied in the literature include device drivers, schedulers, and interrupts, see e.g. [18, 58, 66].

7.3 Side-channel attacks in cryptography

In [85], Kocher presents a practical timing attack on RSA and suggests that many vectors, including the cache, can be exploited to launch side-channel attacks. Aciiçmez and Schindler [6] demonstrate that not only data cache, but instruction cache attacks are also effective.

In 2012, Mowery et al [101] discussed some difficulties in performing cache timing attacks on modern architectures and systems. They claimed that cache timing attacks have become significantly more difficult to perform, due to new technologies such as dedicated hardware for cryptography, multicore processors or prefetching. Further research, such as [13, 75], showed that cache attacks are still very much relevant even on modern architectures.

There are also other timing attacks that do not use the cache as a side-channel, such as the lucky thirteen attack on TLS [10], which uses a timing side-channel to recover plaintexts.

As mentioned before, several countermeasures have been proposed for these attacks, see for example [106, 126, 88, 48, 120, 53]. One relatively recent countermeasure consists of using page coloring to mitigate the cache side-channel [116]. Page coloring is a software-based technique that directs how memory pages are mapped to cache lines. This technique can be used to allocate memory regions that need protecting in a specialized secure color that is not shared with other operating systems, and therefore isolated. The resulting mechanism is somewhat similar to stealth memory. Other countermeasures include specialized prefetching [62], called *disruptive* prefetching; time padding [38]; and obfuscated execution [108].

Over the last decade, researchers have developed abstract models of cryptography that capture side-channels, and developed constructions that are secure in these models, see e.g. [56] for a survey.

7.4 Analysis tools for cache-based attacks

CtGrind[1] is an extension of ValGrind that can be used to check automatically that an implementation is constant-time. It reuses a Valgrind module called *memcheck* that checks if the program ever uses the value of an uninitialized variable. By marking secret data as uninitialized and running memcheck, the tool can detect and report all memory accesses and branches that depend on secret information.

CacheAudit [54] is an abstract-interpretation based framework for estimating the amount of leakage through the cache in straightline x86 executables. CacheAudit has been used to show that several applications do not leak information through the cache and to compute an upper bound for the information leaked through the cache by AES. These guarantees hold for a single run of the program, i.e. in the non-concurrent attacker model. A follow-up [26] provides an upper bound for the leakage of AES in an abtract version of the concurrent attacker model; however, the bound is only valid under strong restrictions, e.g. on scheduling. Moreover, the results of [26] cannot be used to assert the security of constant-time programs against concurrent cache attacks.

7.5 Non-interference

Non-interference originates from the seminal work of Goguen and Meseguer [63, 64], and has been generalized in numerous dimensions. Mantel [96] provides a thorough review of a large body of prior work, and elaborates a general framework to formally specify and verify information flow policies. Building upon [96], von Oheimb [103] revisits and generalizes the classical notion of non-interference for state-based systems presented by Rushby in [111]. In particular, he introduces a notion of non-leakage which focuses on information flow during system runs rather on the observability of actions, and guarantees that secret information present in the initial state of the system does not leak out of the domain(s) it is intended to be confined to. Non-leakage combined with non-interference give rise to the notion of non-influence. One minor shortcoming of non-influence is that it only considers traces that are generated by the same sequence of actions, whereas it is often desirable to prove a refined and stronger property which also consider traces that coincide in the actions performed by the observing domain. Our isolation theorems in Chapter 3 and 5 are stated as non-leakage and non-influence theorems respectively, but avoid the aforementioned shortcoming.

7.6 Language-based protection mechanisms

Many authors have developed language-based protection methods against side-channel attacks. Agat [8] defines an information flow type system that only accepts statements branching on secrets if the branches have the same pattern of memory accesses, and a type-directed transformation to make programs typable. Molnar et al [100] define the program counter model, which is equivalent to path non-interference, and give a program transformation for making programs secure in this model. Coppens et al [48] use selective if-conversion to remove high branches in programs. Zhang et al [131] develop a contract-based approach to mitigate side-channels. Enforcement of contracts

[1] It was developed circa 2010 by Adam Langley and is available from `https://github.com/agl/ctgrind/`.

on programs is performed using a type system, whereas informal analyses are used to ensure that the hardware comply with the contracts. They prove soundness of their approach. However, they do not consider the concurrent attacker model and they do not provide an equivalent of system-level non-interference. Stefan et al [117] also show how to eliminate cache-based timing attacks, but their adversary model is different.

More recently, Liu et al [95] define a type system for an information flow policy called memory-trace non-interference in the setting of oblivious RAM. Their type system has similar motivations as ours, but operates on source code and deals with a different attacker model.

7.7 Verified cryptographic implementations

There is a wide range of methods to verify cryptographic implementations: type-checking, see e.g. [33], deductive verification, see e.g. [55], code generation, see e.g. [40] and model extraction, see e.g. [9]. However, these works do not consider side-channels.

Recently, Almeida et al [12] extend the EasyCrypt framework [25] to reason about the security of C-like implementations in idealized models of leakage, such as the Program Counter Model, and leverage CompCert to carry security guarantees to executable code; moreover they, instrument CompCert with a simple check on assembly programs to ensure that a source C program that is secure in the program counter model is compiled into an x86 program that is also secure in this model.

7.8 Verified compilation and analyses

CompCert [93] is a flagship verified compiler that has been used and extended in many ways; except for [12], these works are not concerned with security. Type-preserving and verifying compilation are alternatives that have been considered for security purposes; e.g. Chen et al [43] and Barthe et al [28] develop type-preserving compilers for information flow.

The generic static analyzer Verasco [76] integrates with the CompCert compiler and establishes the absence of run-time errors of programs. In addition to the proof of soundness of the analyzer, the results are shown to be carried over to the executable code generated by the compiler.

Formal verification of information flow analyses is an active area of research; e.g. Barthe et al [27] and Amtoft et al [15] formally verify type-based and logic-based methods for enforcing information flow policies in programs. More recently, Azevedo et al [17] formally verify a clean-slate design that enforces information flow.

Leroy and Robert [94] have developed a points-to analysis in the CompCert framework. We apply and formalize similar techniques, but combine the alias information with information flow tracking.

Other previous work on mechanized verification of static analyses has been mostly focused on inference of Java bytecode types [84], compiler optimizations [49, 32], control flow analysis [39] or abstract interpretation for value analysis [35].

Chapter 8

Conclusions

In this thesis, we have developed an idealized model of virtualization that includes a specification of the behavior of a virtualization hypervisor, and a very realistic memory model, with virtual memory, cache and TLB. The model captures at an abstract level cache-based leakage in the spirit of leakage resilient cryptography. We have observed that in absence of any specific countermeasure, cache-based side-channels attacks are possible in the model, and proved isolation results that give sufficient conditions to enforce strong isolation between operating systems.

Constant-time cryptography is often advocated as a solution against cache-based attacks. In this work, we have also developed a certified static analysis for constant-time cryptography, and given the first formal proof that constant-time programs are indeed protected against concurrent cache-based attacks.

Moreover, we have extended our analysis to the setting of stealth memory; to this end we have defined a new program classification, which we call S-constant-time, and developed a certified static analysis for this class of programs. Furthermore, we have proved that S-constant-time programs are also protected against cache timing attacks. This constitutes the first formal security analysis of stealth memory.

Our results have been formalized using the Coq proof assistant on top of the CompCert compiler, and our analyses have been validated experimentally on a representative set of cryptographic algorithms.

Table 8.1 shows the specification and proof size of the main parts of our development in lines of Coq code (LoC), as reported by the `coqwc` tool.

	Specification	Proofs	**Total** [1]
Basic model and lemmas	3.2k	6.8k	11.6k
Valid state invariance	180	25.0k	30.0k
Stealth model and theorems	2.1k	8.2k	11.8k
Static analyses and proofs	1.5k	2.0k	4.1k
Interface between model and CompCert	153	20	221
Total	7.1k	42.1k	**57.7k**

Table 8.1: LoC of Coq development

Throughout the development we made extensive use of Coq module system to define clear and effective interfaces between the different parts of the model. We also used Coq automation facilities to some degree, in general following Chlipala's [44] approach to large scale engineering of formal developments. This allowed us to save time when faced with the unavoidable rewriting

[1] The total LoC number includes comments and headers

of theorem statements and definitions. The state equivalence definition, for example, required many iterations in order to obtain strong enough conditions to prove the non-interference result we wanted.

The development of the basic model without stealth memory took us around one year; and included simplifying and rewriting some of the previous definitions [21], and implementing the new VIPT cache. The stealth memory extensions took approximately another year; and consisted of the definition of the new actions and invariants, plus the new state equivalence and proof of the isolation theorem. The static analysis and CompCert integration took around six months; starting from a proof of concept of the analysis for a simple artificial language, then moving to CompCert RTL intermediate language, before finally setting on the Mach analysis presented here.

One direction of future work is to improve our tool by implementing a value analysis to improve the static analysis for constant-time and S-constant-time programs. A strong enough value analysis would allow programs to be checked without modifications.

In addition, our static analysis is performed at the lower levels of the compiling chain, after all compiler optimizations; see [28], for example, for a discussion on how compiler optimizations may break information flow typing. This makes it more difficult for developers to predict whether their applications will pass stealth verification. Another improvement to our current approach is to develop the static analysis closer to the C language and prove that its results are preserved by compilation, in the sense that all accepted programs are compiled into programs that pass stealth verification.

Another direction for future work is to extend our analysis to programs that branch on secrets. This would allow us to analyze a few cryptographic implementations with high branching, such as the modular exponentiation used in some RSA implementations [85]; but more importantly, it will allow us to analyze a broader type of applications that are not cryptographic. Many such applications handle private information and can also be the target of cache timing attacks, such as the recent Oren et al [105] Javascript-based remote attack. One strategy to handle high branching is to extend CompCert with predicated instructions and if-conversion. The work of Barthe *et al* [24] has some preliminary work in this direction.

Regarding the idealized virtualization model, there are many possible extensions. The most interesting ones include modeling transparent page sharing, the instruction cache, IO devices and multi-core. See Sections 2.1.1 and 2.1.2 for an in-depth discussion.

Bibliography

[1] Data encryption standard (DES). Technical Report FIPS PUB 46, Federal Information Processing Standards Publications, 1977.

[2] Advanced encryption standard (AES). Technical Report FIPS PUB 197, Federal Information Processing Standards Publications, 2001.

[3] Secure Hash Standard. Technical Report FIPS PUB 180-4, Federal Information Processing Standards Publications, 2012.

[4] 3rd Generation Partnership Project. Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2; document 2: SNOW 3G specification, 2006.

[5] O. Aciiçmez and Çetin Kaya Koç. Trace-driven cache attacks on AES (short paper). In *8th International Conference on Information and Communications Security (ICICS 2006)*, volume 4307 of *LNCS*, pages 112–121. Springer, 2006.

[6] O. Aciiçmez and W. Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *CT-RSA'08*, volume 4964 of *LNCS*, pages 256–273. Springer, 2008.

[7] O. Aciiçmez, W. Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the AES. In *CT-RSA 2007*, volume 4377 of *LNCS*, pages 271–286. Springer, 2007.

[8] J. Agat. Transforming out Timing Leaks. In *Proceedings POPL'00*, pages 40–53. ACM, 2000.

[9] M. Aizatulin, A. D. Gordon, and J. Jürjens. Computational verification of C protocol implementations by symbolic execution. In *CCS 2012*, pages 712–723. ACM, 2012.

[10] N. Al Fardan and K. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540, May 2013.

[11] E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. Paul. Verifying shadow page table algorithms. In R. Bloem and N. Sharygina, editors, *Formal Methods in Computer-Aided Design, 10th International Conference (FMCAD'10)*, Switzerland, 2010. IEEE CS.

[12] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *CCS 2013*, 2013.

[13] H. Aly and M. ElGayyar. Attacking AES using Bernstein's attack on modern processors. In A. Youssef, A. Nitaj, and A. Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 127–139. Springer Berlin Heidelberg, 2013.

[14] AMD. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[15] T. Amtoft, J. Dodds, Z. Zhang, A. W. Appel, L. Beringer, J. Hatcliff, X. Ou, and A. Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In *POST 2012*, volume 7215 of *LNCS*, pages 369–389. Springer, 2012.

[16] ARM Limited. mbed TLS. See `https://tls.mbed.org/`.

[17] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *POPL 2014*. ACM, 2014.

[18] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In Y. Berbers and W. Zwaenepoel, editors, *EuroSys*, pages 73–85. ACM, 2006.

[19] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[20] G. Barthe, G. Betarte, J. Campo, and C. Luna. Formally verifying isolation and availability in an idealized model of virtualization. In *FM 2011*, pages 231–245. Springer-Verlag, 2011.

[21] G. Barthe, G. Betarte, J. Campo, and C. Luna. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In *CSF 2012*, pages 186–197, 2012.

[22] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1267–1279, New York, NY, USA, 2014. ACM.

[23] G. Barthe, G. Betarte, J. D. Campo, J. M. Chimento, and C. Luna. Formally verified implementation of an idealized model of virtualization. In R. Matthes and A. Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, volume 26 of *LIPIcs*, pages 45–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[24] G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end - static single assignment meets compcert. In *ESOP 2012*, pages 47–66, 2012.

[25] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *CRYPTO 2011*, volume 6841 of *LNCS*, Heidelberg, 2011.

[26] G. Barthe, B. Köpf, L. Mauborgne, and M. Ochoa. Leakage resilience against concurrent cache attacks. In *POST*, 2014.

[27] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *ESOP 2007*, pages 125–140, 2007.

[28] G. Barthe, T. Rezk, and D. A. Naumann. Deriving an information flow checker and certifying compiler for java. In *S&P 2006*, pages 230–242. IEEE Computer Society, 2006.

[29] C. Baumann, H. Blasum, T. Bormer, and S. Tverdyshev. Proving memory separation in a microkernel by code level verification. In W. Steiner and R. Obermaisser, editors, *1st International Workshop on Architectures and Applications for Mixed-Criticality Systems (AMICS 2011)*, Newport Beach, CA, USA, 2011. IEEE Computer Society.

[30] D. Bernstein. Salsa20 specification, 2005.

[31] D. J. Bernstein. Cache-timing attacks on AES, 2005. Available from author's webpage.

[32] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Proc. of TYPES 2006*, volume 3839 of *LNCS*, pages 66–81. Springer, 2006.

[33] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL 2010*. ACM, 2010.

[34] E. Biham. A fast new DES implementation in software. In E. Biham, editor, *Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer Berlin Heidelberg, 1997.

[35] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Proc. of the 20th Static Analysis Symposium (SAS 2013)*, volume 7935 of *LNCS*, pages 324–344. Springer-Verlag, 2013.

[36] A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique cryptanalysis of the full AES. In D. Lee and X. Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 344–371. Springer Berlin Heidelberg, 2011.

[37] J. Bonneau and I. Mironov. Cache collision timing attacks against AES. In *CHES '06*, 2006.

[38] B. A. Braun, S. Jana, and D. Boneh. Robust and efficient elimination of cache and timing side channels. *CoRR*, abs/1506.00189, 2015.

[39] D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[40] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations. In *ARES 2012*, pages 65–74. IEEE Computer Society, 2012.

[41] A. Canteaut, C. Lauradoux, and A. Seznec. Understanding cache attacks. Rapport de recherche RR-5881, INRIA, 2006.

[42] T. Chardin, P.-A. Fouque, and D. Leresteux. Cache timing analysis of RC4. In *ACNS 2011*, volume 6715 of *LNCS*, pages 110–129, 2011.

[43] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI 2010*, pages 412–423. ACM, 2010.

[44] A. Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[45] J. Chrząszcz. Implementing modules in the Coq system. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer Berlin Heidelberg, 2003.

[46] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[47] E. Cohen. Validating the Microsoft hypervisor. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM '06*, volume 4085 of *LNCS*, pages 81–81. Springer, 2006.

[48] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P 2009*, pages 45–60, 2009.

[49] S. Coupet-Grimal and W. Delobel. A uniform and certified approach for two static analyses. In *Proc. of TYPES 2004*, volume 3839 of *LNCS*, pages 115–137, 2004.

[50] J. Daemen, J. Daemen, J. Daemen, V. Rijmen, and V. Rijmen. AES proposal: Rijndael, 1998.

[51] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *CCS 2013*, pages 223–234, 2013.

[52] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[53] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4):35:1–35:21, Jan. 2012.

[54] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Usenix Security 2013*, 2013.

[55] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols. In *CSF 2011*, pages 3–17. IEEE Computer Society, 2011.

[56] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *FOCS*, pages 293–302. IEEE Computer Society, 2008.

[57] U. Erlingsson and M. Abadi. Operating system protection against side-channel attacks that exploit memory latency. Technical Report MSR-TR-2007-117, Microsoft Research, 2007.

[58] X. Feng, Z. Shao, Y. Guo, and Y. Dong. Certifying low-level programs with hardware interrupts and preemptive threads. *J. Autom. Reasoning*, 42(2-4):301–347, 2009.

[59] P.-A. Fouque, J. Jean, and T. Peyrin. Structural evaluation of AES and chosen-key distinguisher of 9-round AES-128. In R. Canetti and J. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 183–203. Springer Berlin Heidelberg, 2013.

[60] J. Franklin, S. Chaki, A. Datta, J. McCune, and A. Vasudevan. Parametric verification of address space separation. In P. Degano and J. Guttman, editors, *Proceedings of POST'12*, volume 7215 of *LNCS*, 2012.

[61] J. Franklin, S. Chaki, A. Datta, and A. Seshadri. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *IEEE Symposium on Security and Privacy*, pages 365–379. IEEE Computer Society, 2010.

[62] A. Fuchs and R. B. Lee. Disruptive prefetching: Impact on side-channel attacks and cache designs. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 14:1–14:12, New York, NY, USA, 2015. ACM.

[63] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[64] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

[65] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7:34–45, June 1974.

[66] A. Gotsman and H. Yang. Modular verification of preemptive OS kernels. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 404–417. ACM, 2011.

[67] D. Greve, M. Wilding, and W. M. V. Eet. A separation kernel formal security policy. In *Proc. Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.

[68] D. Gullasch, E. Bangerter, and S. Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *S&P 2011*, pages 490–505, 2011.

[69] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, Oct. 2014. USENIX Association.

[70] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 346–355, NY, USA, 2006. ACM.

[71] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[72] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

[73] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on ARM: System virtualization using xen hypervisor for ARM-based secure mobile phones. In *5th IEEE Consumer and Communications Networking Conference*, 2008.

[74] G. Irazoqui, M. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! a fast, cross-VM attack on AES. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688 of *Lecture Notes in Computer Science*, pages 299–319. Springer International Publishing, 2014.

[75] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Fine grain cross-VM attacks on Xen and VMware are possible! *IACR Cryptology ePrint Archive*, 2014:248, 2014.

[76] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *POPL 2015: 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, Mumbai, India, Jan. 2015. ACM.

[77] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[78] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. *Journal of Computer Security*, 8(2–3):141–158, 2000.

[79] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[80] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security 2012*, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[81] G. Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, 2009.

[82] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.

[83] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP 2009*, pages 207–220. ACM, 2009.

[84] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*, 28(4):619–695, 2006.

[85] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[86] F. Koeune and J.-J. Quisquater. A timing attack against Rijndael. Technical report, Université Catholique de Louvain, 1999.

[87] R. Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, University of New South Wales, 2011.

[88] J. Kong, O. Aciiçmez, J. Seifert, and H. Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*, pages 393–404. IEEE Computer Society, 2009.

[89] M. N. Krohn and E. Tromer. Noninterference for a practical DIFC-based operating system. In *IEEE Symposium on Security and Privacy*, pages 61–76. IEEE Computer Society, 2009.

[90] M. N. Krohn, A. Yip, M. Z. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 321–334. ACM, 2007.

[91] G. Leander, E. Zenner, and P. Hawkes. Cache Timing Analysis of LFSR-Based Stream Ciphers. In *IMACC 2009*, volume 5921 of *LNCS*, pages 433–445. Springer, 2009.

[92] D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In A. Cavalcanti and D. Dams, editors, *FM 2009*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009.

[93] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL 2006*, pages 42–54. ACM, 2006.

[94] X. Leroy and V. Robert. A formally-verified alias analysis. In *CPP*, pages 11–26, 2012.

[95] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *CSF 2013*, pages 51–65, 2013.

[96] H. Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Univ. des Saarlandes, 2003.

[97] W. Martin, P. White, F. Taylor, and A. Goldberg. Formal construction of the mathematically analyzed separation kernel. In *The Fifteenth IEEE International Conference on Automated Software Engineering*, 2000.

[98] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[99] S. Micali and L. Reyzin. Physically observable cryptography (extended abstract). In *TCC 2004*, pages 278–296, 2004.

[100] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *ICISC 2005*, pages 156–168, 2005.

[101] K. Mowery, S. Keelveedhi, and H. Shacham. Are AES x86 cache timing attacks still feasible? In *Proceedings of CCSW '12*. ACM, 2012.

[102] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. G., and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *S&P 2013*, pages 415–429, 2013.

[103] D. v. Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004*, volume 3193 of *LNCS*, pages 225–243. Springer, 2004.

[104] D. v. Oheimb, V. Lotz, and G. Walter. Analyzing SLE 88 memory management security using Interacting State Machines. *International Journal of Information Security*, 4(3):155–171, 2005.

[105] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox - practical cache attacks in Javascript. *CoRR*, abs/1502.07373, 2015.

[106] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2006.

[107] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel, 2002.

[108] A. Rane, C. Lin, and M. Tiwari. Raccoon: Digital side-channel freedom through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., Aug. 2015. USENIX Association.

[109] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud! Exploring information leakage in third-party compute clouds. In *CCS 2009*, pages 199–212. ACM Press, 2009.

[110] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[111] J. M. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, 1992.

[112] B. Schneier. The Blowfish encryption algorithm.

[113] B. Schneier. The Blowfish source code.

[114] T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In *ITP 2011*, Nijmegen, The Netherlands, 2011.

[115] Z. Shao. Certified software. *Commun. ACM*, 53(12):56–66, 2010.

[116] J. Shi, X. Song, H. Chen, and B. Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on*, pages 194–199, June 2011.

[117] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 718–735. Springer, 2013.

[118] H. Tews, M. Völp, and T. Weber. Formal memory models for the verification of low-level operating-system code. *J. Autom. Reasoning*, 42(2-4):189–227, 2009.

[119] M. Tiwari, J. Oberg, X. Li, J. Valamehr, T. E. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In R. Iyer, Q. Yang, and A. González, editors, *ISCA*, pages 189–200. ACM, 2011.

[120] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[121] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *CHES 2003*, volume 2779 of *LNCS*, pages 62–76. Springer, 2003.

[122] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[123] P. Varanasi and G. Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, New York, NY, USA, 2011. ACM.

[124] VMWare. Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing. `http://kb.vmware.com/selfservice/search.do?cmd=displayKC&docType=kc&docTypeID=DT_KB_1_1&externalId=2080735`. Accessed May 2015.

[125] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA 2007*, pages 494–505. ACM, 2007.

[126] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008), November 8-12, 2008, Lake Como, Italy*, pages 83–93. IEEE Computer Society, 2008.

[127] D. Wheeler and R. Needham. TEA, a tiny encryption algorithm. In B. Preneel, editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 363–366. Springer Berlin Heidelberg, 1995.

[128] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of PLDI'10*, pages 99–110. ACM, 2010.

[129] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In K. Fu and J. Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732. USENIX Association, 2014.

[130] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, pages 263–278. USENIX Association, 2006.

[131] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *CCS 2011*, pages 563–574. ACM, 2011.