Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique »

présentée et soutenue publiquement par

Clémentine MAURICE

le 28 octobre 2015

Fuites d'information dans les processeurs récents et applications à la virtualisation

Directeurs de thèse :	**Aurélien FRANCILLON**
	Christoph NEUMANN
Co-encadrement de la thèse :	**Olivier HEEN**

Jury

M. Thomas JENSEN, Directeur de recherche, Inria Rennes, France	Président de jury
M. Gildas AVOINE, Professeur, IRISA et INSA Rennes, France	Rapporteur
M. Jean-Pierre SEIFERT, Professeur, TU Berlin, Allemagne	Rapporteur
M. Stefan MANGARD, Professeur, TU Graz, Autriche	Examinateur
M. Pierre PARADINAS, Professeur, CNAM, Paris, France	Examinateur

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

T H È S E

Information Leakage on Shared Hardware
Evolutions in Recent Hardware and Applications to Virtualization

Clémentine Maurice

A doctoral dissertation submitted to:
Télécom ParisTech

Advisors:
Aurélien Francillon, Eurecom, Sophia Antipolis, France
Christoph Neumann, Technicolor, Rennes, France
Olivier Heen, Technicolor, Rennes, France

Jury:

Thomas Jensen, Inria Rennes, France	President
Gildas Avoine, IRISA and INSA Rennes, France	Reviewer
Jean-Pierre Seifert, TU Berlin, Germany	Reviewer
Stefan Mangard, TU Graz, Austria	Examiner
Pierre Paradinas, CNAM, Paris, France	Examiner

October 2015

Contents

Résumé

Dans un environnement virtualisé, l'hyperviseur fournit l'isolation au niveau logiciel, mais l'infrastructure partagée rend possible des attaques au niveau matériel. Les attaques par canaux auxiliaires ainsi que les canaux cachés sont des problèmes bien connus liés aux infrastructures partagées, et en particulier au partage du processeur. Cependant, ces attaques reposent sur des caractéristiques propres à la microarchitecture qui change avec les différentes générations de matériel. Ces dernières années ont vu la progression des calculs généralistes sur processeurs graphiques (aussi appelés GPUs), couplés aux environnements dits *cloud*. Cette thèse explore ces récentes évolutions, ainsi que leurs conséquences en termes de fuites d'information dans les environnements virtualisés. Premièrement, nous investiguons les microarchitectures des processeurs récents. Notre première contribution est C5, un canal caché sur le cache qui traverse les coeurs d'un processeur, évalué entre deux machines virtuelles. Notre deuxième contribution est la rétro-ingénierie de la fonction d'adressage complexe du dernier niveau de cache des processeurs Intel, rendant la classe des attaques sur les caches facilement réalisable en pratique. Finalement, dans la dernière partie nous investiguons la sécurité de la virtualisation des GPUs. Notre troisième contribution montre que les environnements virtualisés sont susceptibles aux fuites d'informations sur la mémoire d'un GPU.

Abstract

In a virtualized environment, the hypervisor provides isolation at the software level, but shared infrastructure makes attacks possible at the hardware level. Side and covert channels are well-known issues of shared hardware, and in particular shared processors. However, they rely on microarchitectural features that are changing with the different generations of hardware. The last years have also shown the rise of General-Purpose computing on Graphics Processing Units (GPGPU), coupled to so-called *cloud* environments. This thesis explores these recent evolutions and their consequences in terms of information leakage in virtualized environments. We first investigate the recent processor microarchitectures. Our first contribution is C5, a cross-core cache covert channel, evaluated between virtual machines. Following this work, our second contribution is the reverse engineering of the complex addressing function of the last-level cache of Intel processors, rendering the class of cache attacks highly practical. In the last part, we investigate the security of GPU virtualization. Our third contribution shows that virtualized environments are susceptible to information leakage from the GPU memory.

Acknowledgments

Many friends, family members and colleagues supported me during my thesis. I am glad I know them, and these few words cannot express all my gratitude to them.

I am grateful to my thesis committee Thomas Jensen, Gildas Avoine, Jean-Pierre Seifert, Stefan Mangard and Pierre Paradinas, for accepting to come to Rennes for my defense and for their constructive questions and comments. I particularly thank Jean-Pierre Seifert and Gildas Avoine for taking the time to read the manuscript and providing feedback.

I sincerely thank my thesis advisors Aurélien Francillon, Olivier Heen and Christoph Neumann, who, contrary to what the formal title page says, equally advised my thesis. You gave me a great balance between guidance and freedom in my research, and also trusted me with my crazy cache attack ideas. Thank you for your patience, support and encouragements, especially during rough times. I really enjoyed working with you!

My time in Technicolor would not have been as enjoyable without my former and current colleagues. Thanks for the fruitful technical discussions or fun conversations over coffee and for the general good mood. Special thanks to Fabien for sharing joy, despair and rooibos, and good luck for the end of the ride! Also thanks to the friendly S3 team of Eurecom for receiving me for my short and sunny stays. Best wishes to all the former and current members.

I would like to thank Stefan Mangard who invited me at TU Graz, as well as the whole Secure Systems team for their warm welcome. I really enjoyed my stay there, and am happy to come back. I especially thank Daniel — it's been great working with you and I am glad we continue to do so. Also thanks for accepting to read parts of this manuscript. Many thanks to Maria and Raphael as well, who coped with a great deal of enthusiasm out of a few bit flips.

A lot of thanks to the great people that are my friends and family. My parents Françoise and Patrick provided a computer-friendly environment that filled my nerdy aspirations during my childhood and supported me during my studies and this thesis. My extended family, including my in-laws Soizik and Marc, was always kind and encouraging. My friends, whether working on their own theses or observing this strange world from outside, were incredibly supportive. You're too many to be listed here, but I feel lucky to have y'all in my life. Thanks for your support and help, and all the good times shared together.

Last but not least, I would like to thank my beloved husband, Erwan. Your constant support over the years has been exceptionally important to me. I never could have done it without you, and for this I am forever grateful. To many more adventures!

Foreword

This manuscript presents the work realized during my CIFRE PhD at Technicolor and Eurecom. It is mainly based on the published articles [MNHF14, MNHF15, MLSN^{+}15] for which I am the main author, and the pre-print [GMM15] written as a follow-up of my visit at TU Graz, for which I am a co-author. During my stay at Technicolor, I also contributed to the published article [MON^{+}13] that is out of the scope of this thesis.

List of publications

International conferences

Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel's Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, November 2015.

Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'15)*, July 2015. Best paper award.

Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Confidentiality Issues on a GPU in a Virtualized Environment. In *Proceedings of the 18th International Conference on Financial Cryptography and Data Security (FC'14)*, March 2014.

Clémentine Maurice, Stéphane Onno, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Improving 802.11 Fingerprinting of Similar Devices by Cooperative Fingerprinting. In *Proceedings of the 2013 International Conference on Security and Cryptography (SECRYPT'13)*, August 2013.

Pre-print

Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *arXiv:1507.06955v1*, July 2015.

Presentations

Clémentine Maurice. Information Leakage in Virtualized Environments: A Journey into GPU Memory and CPU Caches. Secure Systems group seminar, IAIK, TU Graz, Austria, June 2015.

Clémentine Maurice, Christoph Neumann, Olivier Heen, Aurélien Francillon. Information Leakage in Virtualized GPUs. Poster presented at Security and Privacy thematic day of Labex UCN, Sophia Antipolis, France, December 2013.

Clémentine Maurice, Christoph Neumann, Olivier Heen, Aurélien Francillon. Information Leakage in a GPU in a Virtualized Environment. Presentation at the Third Workshop on Storage and Cloud Computing, Rennes, France, November 2013.

Awards and grants

RAID student travel grant, November 2015.

DIMVA Best paper award, July 2015.

Funding by the COST Cryptacus to visit TU Graz for a Short Term Scientific Mission, June 2015.

Hardware donation of a GPU by NVIDIA, April 2013.

1

Introduction

Contents

1.1 Context

Cloud computing was introduced this last decade and gained in popularity ever since. It provides customers – users or enterprises – on-demand solutions for computing and storage in dedicated datacenters. For customers, the benefit is simplicity: the same services run on different hardware platforms, without having to consider the specificities of hardware. It also offloads the need to manage an infrastructure to a dedicated provider. For cloud providers, the benefit is cost efficiency: several virtual machines, that can be owned by different customers, run on the same physical machine. Cloud computing relies heavily on *virtualization*, which consists in the decoupling of software services from underlying hardware. Hardware sharing is a central aspect of the cloud computing environment. Among others, two important pieces of hardware are virtualized today: the Central Processing Unit (CPU), and the Graphics Processing Unit (GPU).

The CPU is the computing part of a computer system. As predicted by Moore's law [Moo75], x86 processors that constitute personal computers and servers become smaller, faster, and more power-efficient. The speed of the processor has increased exponentially over the last decades. Modern CPUs are composed of several cores, that are basic processing units. They also use *caches*, that are small but fast memories filling the performance gap between the processor and the main memory.

The GPU is used primarily in personal computers for multimedia applications such as video gaming or high-definition video editing. As these new multimedia usages grew, GPUs became widespread. Their highly parallel architecture changed towards general purpose computing. Today, GPUs are also used for high performance computing, including in servers and clusters. As a result, they made an appearance in services offered by cloud computing providers.

Shared hardware between different tenants induces the potential threat of information leakage. In particular, the CPU can be accessed concurrently by different users, and the CPU cache is heavily shared. This leads to *covert* and *side channels*. The GPU is time-shared, *i.e.*, not two users can access it at the same time, but they can access it one after the other. Memory isolation is crucial in this case to prevent information leakage. Attacks on these systems either aim to actively exchange information between two processes, or for a spy process to exfiltrate secret information from a victim process.

1.2 Problem statement

Information leakage due to shared hardware is a known problem and has been extensively studied. However, these attacks are highly dependent on the hardware. Yet, we see evolutions both in hardware design and adoption. First, CPU microarchitectures change frequently. For example, Intel has built a new one nearly every year since 2009. Second, GPUs have been designed to provide maximum performance and not for concurrent accesses and security. Nonetheless, they have been recently offered by several cloud computing providers. Due to the evolution of hardware or to the countermeasures in production, some attacks are rendered more complex or impossible to perform. In contrast, some modifications are carried out with performance in mind, which is often conflicting with security. We therefore question: *How do the recent evolutions impact information leakage due to hardware sharing?*

This question gives rise to some challenges concerning the investigation of these security issues. Indeed, we are faced with two layers of obscurity.

The first is the cloud provider. For reasons as diverse as security concerns or business model, cloud providers are reluctant to give details about their infrastructure. The second is the hardware itself. As it is increasingly complex, both CPUs and GPUs are built with some performance-critical parts that remain undocumented.

1.3 Contributions

The goal of this thesis is to study the impact of evolutions in recent hardware in terms of information leakage on shared hardware. We also apply our findings to virtualized environments which are widely used today, and which constitute a natural use case of hardware sharing. This thesis presents the work done during my PhD and makes contributions along three axes.

Building a Cross-Core Cache Covert Channel Covert channels were demonstrated to violate isolation and, typically, allow data exfiltration. Several covert channels have been proposed that rely on the CPU cache. However, these covert channels are either slow or impractical due to the *addressing uncertainty*. This uncertainty is caused by the additional layer of indirection in virtualized environment, and by the addressing mode of recent last-level caches. Using shared memory would elude addressing uncertainty, but shared memory is not available in most practical setups. We build C5, a covert channel that tackles addressing uncertainty without requiring any shared memory, making this covert channel fast and practical. We are able to transfer messages on modern hardware across any cores of the same processor. The C5 covert channel targets the last-level cache that is shared across all cores. It exploits the inclusive feature of caches, allowing a core to evict lines in the private first level cache of another core. We experimentally evaluate our covert channel in native and virtualized environments. In particular, we successfully establish a covert channel between virtual machines running on different cores. We measure a bitrate one order of magnitude above previous cache-based covert channels in the same setup.

Reverse-engineering the complex addressing function of Intel last-level cache The last-level cache of recent processors is split in *slices*. While predicting the slice used by an address is simple in older processors, recent processors are using an undocumented technique called *complex addressing*. This renders some attacks more difficult and makes other attacks impossible, because of the loss of precision in the prediction of cache collisions. Previous work only manually reverse-engineered the complex addressing function of one specific processor. We build an automatic and generic method for reverse-engineering Intel complex addressing, consequently rendering the class of cache attacks highly practical. Our method relies on CPU hardware performance counters to determine the cache slice an address is mapped to. We show that our method gives a more precise description of the complex addressing function than previous work. We validated our method by reversing the complex addressing function on a diverse set of Intel processors. This set encompasses Sandy Bridge, Ivy Bridge and Haswell microarchitectures, with different number of cores, for mobile and server range processors. We show that knowing the complex addressing improves C5 by several orders of magnitude. We also discuss how knowing the complex addressing function of a cache enables other attacks, such as sandboxed Rowhammer.

Information leakage on GPU memory in virtualized environments Few studies have been conducted on the security implications of General-Purpose Computing on Graphics Processing Units (GPGPU) combined to cloud computing. Our objective is to highlight possible information leakage due to GPUs in virtualized and cloud computing environments. We provide insight into the different GPU virtualization techniques, along with their security implications. We systematically experiment and analyze the behavior of GPU global memory in the case of direct device assignment. We find that GPU global memory is zeroed only in some configurations, as a side effect of Error Correction Codes (ECC) and not for security reasons. As a consequence, an attacker can recover data of a previously executed GPGPU application in a variety of situations. These situations include setups where the attacker launches a virtual machine after the victim's virtual machine using the same GPU, thus bypassing the isolation mechanisms of virtualization. Memory cleaning is not implemented by the GPU card itself and we cannot generally exclude the possibility of data leakage in cloud computing environments. Furthermore, we discuss possible countermeasures for current GPU clouds users and providers. To the best of our knowledge, this is the first work on information leakage of GPU memory in virtualized environments.

Our contributions in the domain of CPU cache covert channels make a significant advancement in two directions. First, by building a covert channel, we analyze the root causes of the interferences that make information leakage possible in recent processors. Second, the reverse engineering of Intel last-level cache complex addressing function tackles one of the challenges that is the lack of documentation of hardware. Finally, our contribution in the domain of GPU memory isolation paves the way to secure virtualization of GPUs.

1.4 Organization of the thesis

This thesis is organized as follows.

Chapter 2 reviews technical background and the state of the art. It first covers the architecture and virtualization of x86 systems. It also includes information leakage on shared resources such as the memory bus and the CPU, with a focus on CPU caches and GPU memory.

Chapter 3 presents C5, a new covert channel on inclusive last-level caches. This covert channel takes into account the evolution of recent CPUs. We evaluated its bitrate and error rate on setups across cores and across virtual machines.

Chapter 4 details how we reverse-engineered the complex addressing function of recent Intel last-level caches. We evaluated our approach by retrieving the function on a large set of different processors. We also show some security applications that ensued this work.

Chapter 5 documents the impact of GPU virtualization techniques on security. We systematically investigate information leakage on GPU memory, in particular in virtualized environments. We detail two methods to access GPU memory, that require different levels of privileges.

Chapter 6 concludes and gives perspective on future work.

2

State of the art

Contents

This chapter first introduces the architecture of an x86 system (Section 2.1), and in particular the internals of the CPU cache. We then detail hardware virtualization, concerning the CPU, memory and I/O devices, and the different threats in terms of security (Section 2.2). We next present covert and side channels on shared resources (Section 2.3), with a focus on the case of data cache (Section 2.4). Finally, we discuss information leakage on GPU memory (Section 2.5).

2.1 Architecture of an x86 system

2.1.1 CPU

A CPU is a component with one or more processing units called cores. Cores execute the sequence of instructions formed by programs. In this thesis, we focus on the Instruction Set Architecture (ISA) x86, that is present in most

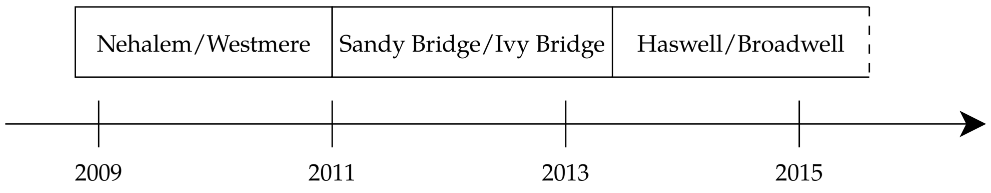

Figure 2.1: Timeline of the different Intel microarchitectures since Nehalem to the most recent one Broadwell.

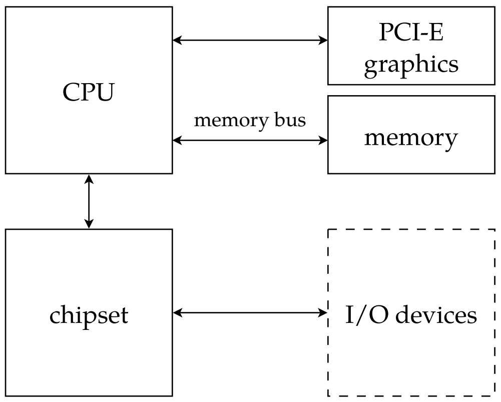

Figure 2.2: Baseline architecture of an x86 system.

personal computers and servers. The ISA can be implemented by a variety of *microarchitectures*. In this thesis, we focus on Intel processors as Intel has a dominant position in market share [For14], compared to AMD. Still, most of this discussion applies also to other x86 processors. Figure 2.1 shows the timeline of the most recent x86 Intel microarchitectures. Figure 2.2 illustrates the baseline architecture and different components of an x86 system.

Major microarchitectural advances in recent processors in terms of performance comprise: pipelining, out-of-order execution, speculative execution, multiple cores and threads, prefetching and caching. In pipelined processors, several instructions can be processed simultaneously, *i.e.*, in a single cycle, at different stages. This increases instruction-level parallelism. In out-of-order processors, the instructions can be processed in an order that is different from the one in the binary, depending on the availability of input data. With speculative execution, the processor performs a task before it is known if the task is actually needed. Intel started manufacturing multi-core processors in 2006; they are now widespread.

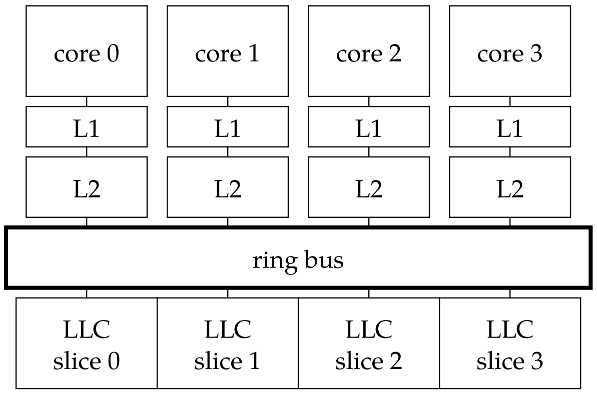

Figure 2.3: Cache hierarchy on Intel processors since Nehalem microarchitecture to the most recent one Broadwell.

We now describe in more detail cache organization, as well as branch prediction.

2.1.1.1 Data cache

Data caches store recently-used data in a fast but small Static Random Access Memory (SRAM). They exploit the concepts of temporal and spatial locality: if some resource is accessed, it is likely to be re-accessed in the near future (temporal), as well as its neighbor resources (spatial).

Cache hierarchy Intel processors use a cache hierarchy similar to the one depicted in Figure 2.3 since the Nehalem microarchitecture and until the most recent Broadwell microarchitecture [Int14a]. There are usually three cache levels, called L1, L2 and L3. The levels L1 and L2 are private to each core, and store several kilobytes. The L3 cache is also called Last-Level Cache (LLC). It is shared among cores and can store several megabytes.

To read or write data in main memory, the CPU first checks the memory location in the L1 cache. If the address is found, it is a *cache hit* and the CPU immediately reads or writes data in the cache line. Otherwise, it is a *cache miss* and the CPU searches for the address in the next level, and so on, until reaching main memory. A cache hit is significantly faster than a cache miss.

Particularities of the last-level cache in recent processors In recent Intel processors, the last-level cache is divided into slices that are connected to the cores through a ring interconnect. Moreover, the last-level cache is inclusive,

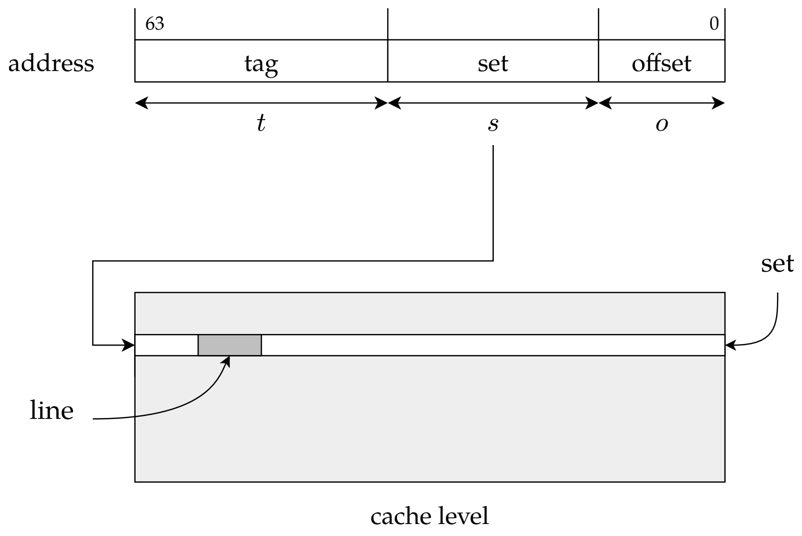

Figure 2.4: Simple addressing scheme.

which means that it is a superset of the L1 and L2, *i.e.*, it contains all data present in L1 and L2. This property does not fully exploit the total available capacity of the cache levels, however, it is an advantageous design for performance reasons, as only one level needs to be checked to know if a line is cached. Inclusiveness also simplifies the cache coherence protocol. To guarantee the inclusion property, a line evicted from the last-level cache is also removed (invalidated) in the L1 and L2 caches.

Addressing scheme Data is transferred between the cache and the memory in 64-byte blocks called *lines*. The location of a particular line depends on the cache structure. Today's caches are *n-way associative*, which means that a cache is composed of sets of n lines. A line is loaded in a specific set depending on its address, and occupies any of the n lines.

With caches that implement a *direct addressing* scheme, memory addresses can be decomposed in three parts: the tag, the set and the offset in the line. The lowest o bits determine the offset in the line, with: $o = \log_2(\text{line size})$. The next s bits determine the set, with: $s = \log_2(\text{number of sets})$. And the remaining t bits form the tag. Figure 2.4 illustrates this scheme.

In contrast to direct addressing, some caches implement a *complex addressing* scheme, where potentially all address bits are used to index the cache. Indeed, in the last-level cache each physical address is associated with a slice via a function that is not documented by Intel, to the best of our knowledge.

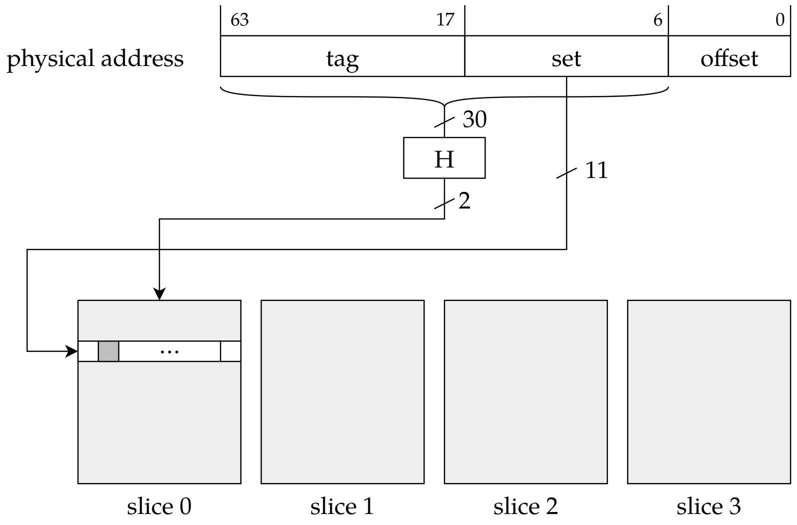

Figure 2.5: Complex addressing scheme on the LLC. This assumes a quad-core processor, and the following characteristics of the LLC: 64B lines and 2048 sets per slice.

As each slice has a cache pipeline, the addressing function is designed to distribute evenly the traffic across all slices for a wide range of memory access patterns, to increase performance. The set is then directly addressed. Intel started implementing this complex addressing scheme on the Sandy Bridge microarchitecture, and onwards (see Table 2.1). Figure 2.5 illustrates this scheme.

The address used to compute the cache location can be either a physical or a virtual address. A *Virtually Indexed, Virtually Tagged* (VIVT) cache only uses virtual addresses to locate data in the cache. Modern processors involve physical addressing, thus cache levels are either *Virtually Indexed Physically Tagged* (VIPT), or *Physically Indexed Physically Tagged* (PIPT). The physical address is not known by processes, thus a process cannot know the location of a specific line for physically indexed caches. Typically, the L1 cache is VIPT, and L2 and L3 are PIPT.

Replacement policy When a cache set is full, a cache line needs to be evicted before storing a new cache line. When a line is evicted from L1 it is stored back to L2, which can lead to the eviction of a new line to the last-level cache, etc. The replacement policy decides the victim line to be evicted. Good replacement policies choose the line that is the least likely to be reused. Used policies include

Table 2.1: Characteristics of the recent Intel microarchitectures.

	Nehalem	**Sandy Bridge**	**Ivy Bridge**	**Haswell**
LLC slices	✓	✓	✓	✓
LLC complex addr.	✗	✓	✓	✓
Replacement policy	LRU	LRU	Quad-Age	Quad-Age

Pseudo Random, Least Recently Used (LRU), and variations of LRU [JTSE10] (see Table 2.1). An adaptive policy can also be used, where the processor dynamically changes the replacement policy depending on the miss rate of specific cache sets [QJP$^+$07]. An efficient replacement policy minimizes the number of cache misses and is thus crucial for performance. These policies are therefore not well documented in recent processors. For instance, the replacement policy used in the Ivy Bridge microarchitecture, a variation of LRU called Quad-Age, only appears as a part of an Intel presentation [JGSW], and is not fully documented to the best of our knowledge. Details of the replacement policies can however be partially reverse-engineered using micro-benchmarks, as it has been done for the Ivy Bridge microarchitecture [Won13].

2.1.1.2 Instruction cache

Similarly to the data cache, the instruction cache stores the most recently used program instructions, and benefits from temporal and spatial locality.

Recent Intel processors have an instruction cache that is private to each core and separate from the data cache for the L1. The L2 and L3 caches are unified, which means that they contain both data and instructions. All considerations we discussed for data caches apply to the instruction cache as well.

2.1.1.3 Branch prediction unit

Two-way branches, e.g., an if-then-else structure, is usually implemented with a conditional jump instruction. The conditional jump can either be “taken” or “not taken”. This cannot be known for sure until the condition has been calculated, which results in wasted cycles in the pipeline if the processor is stalling, waiting for it. Instead, speculative execution is used to improve performance.

The branch predictor predicts the outcome of the branch instructions, using past behavior. Thereby, the processor speculatively continues to execute instructions on the predicted path, without waiting for the outcome to be computed. When a misprediction happens the pipeline is flushed, thus all the speculative instructions have to be dumped and the execution has to start over, resulting in a longer execution time. The penalty depends on the length of the pipeline: the deeper the pipeline, the higher the penalty. Modern processors tend to have longer pipelines, thus the branch predictor is a critical element of modern pipelined processors.

If a branch is predicted to be taken, the next instruction needs to be fetched and issued. This requires knowing the address of this instruction. For that, the CPU uses a Branch Target Buffer, that is the buffer of the target addresses of previously executed branches. The Branch Target Buffer is a cache.

2.1.2 Memory

The main memory (also referred simply as *memory*) is a volatile type of storage, *i.e.*, information is lost when memory is powered off, typically made of Dynamic Random Access Memory (DRAM). Contrary to external memory, e.g., hard disk drives, the main memory is directly accessible by the CPU, via the memory bus.

The memory bus connects the main memory to the memory controller. In recent CPUs, the memory controller has been moved from the motherboard to the CPU itself. It is called an integrated memory controller. The memory bus is actually composed of two different buses: the address bus and the data bus. The CPU sends the address on the address bus, to indicate the location of the requested data, and read or written data is sent via the data bus.

Modern CPUs implement virtual memory using a Memory Management Unit (MMU), a hardware unit between the CPU and the DRAM that maps virtual addresses used by processes to physical addresses.

2.2 Hardware virtualization

Virtualization is the decoupling of software services from the underlying hardware. This solution has gained attraction, with the rise of the *cloud computing*. For users, the benefit is simplicity: the same services run on different hardware platforms, without having to consider the specificities of the hardware. For cloud providers, the benefit is cost efficiency: several Virtual Machines (VMs), that can be owned by different tenants, run on the same physical machine.

Virtualized environment have three main components: the *hypervisor* that is the abstraction layer, a *host operating system* that has privileged access to hardware, and *guest operating systems* that are unprivileged virtual machines. There are two main types of hardware virtualization. *Type 1* hypervisors, also called native or bare-metal, run directly on top of hardware (see Figure 2.6).

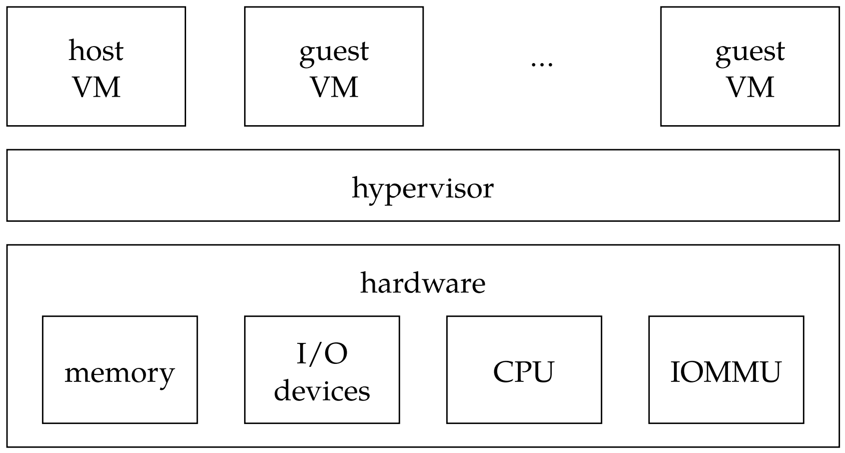

Figure 2.6: Type 1 virtualization.

They manage all the resources, including the CPU, memory and Input/Output (I/O) devices. Examples of type 1 hypervisors include Xen [BDF+03], KVM [KKL+07] and VMware ESXi. *Type 2* hypervisors, also called hosted, run as a process on top of a host operating system. They manage access to hardware from the guest operating systems via calls to the host operating system. Examples of type 2 hypervisors include VMware Player and VirtualBox. In both types of virtualization, the crucial role of the hypervisor is to isolate the different guest operating systems from each other. Type 1 virtualization is the most used in multi-tenants environments such as in cloud computing environments. This is the type of virtualization we focus on in the remainder of this thesis.

2.2.1 CPU

The x86 architecture provides some challenges to virtualization. Indeed, some sensitive instructions do not have the same semantics when executed in Ring 3 (least privileged, user-level application) and in Ring 0 (most privileged, kernel-level).

Software-based virtualization techniques include binary translation and para-virtualization. In *binary translation*, the hypervisor translates the virtual machine instructions before their execution. It replaces non-virtualizable instructions by instructions that behave the same way, but that do not have the same effect on the underlying hardware. Hardware is fully virtualized and the operating system that runs on top of it does not have to be adapted.

To increase performance, a combination of binary translation for kernel-level code and direct execution for user-level code can be used. In *para-virtualization*, a virtualization layer that is similar to the underlying hardware is presented to the virtual machines. The operating system kernel is modified to replace non-virtualizable instructions by hypercalls (requests of privileged operations from a domain to the hypervisor).

Hardware-based virtualization was introduced by Intel and AMD in 2005 and 2006 – respectively named Intel VT-x and AMD-V. It extends the instruction set and creates a new ring for the hypervisor below Ring 0. It allows running an unmodified guest operating system that is unaware of the virtualization layer. When the processor encounters a privileged instruction, it *exits* from the guest mode, lets the hypervisor emulate the instruction, and then returns to the guest mode.

2.2.2 Memory

Memory virtualization handles an additional layer of indirection caused by the MMU: from the guest virtual pages, to the guest physical pages, to the actual machine pages (see Figure 2.7). The hypervisor is responsible to map the guest physical memory to the actual machine memory. To avoid performance penalty, software and hardware methods remove this additional layer of indirection.

With a software-based solution, the hypervisor maintains a shadow page table to directly map the guest virtual memory to the machine memory. However, the monitoring of the guest pages by the hypervisor entails a significant overhead. Indeed, the entries of the shadow page table are built at every guest page fault, which causes a hypervisor exit.

Intel and AMD brought hardware support for MMU virtualization, respectively named EPT (Extended Page Tables) and RVI (Rapid Virtualization Indexing). The guest continues to maintain guest virtual pages to guest physical pages mapping, but the guest physical pages to machine pages mapping is maintained by the hypervisor and exposed to hardware. The physical CPU walks the two tables and caches the guest virtual pages to machine pages mapping in the Translation Lookaside Buffer (TLB). Performance gain is evaluated to more than 40% compared to the software table [VMw09a, VMw09b].

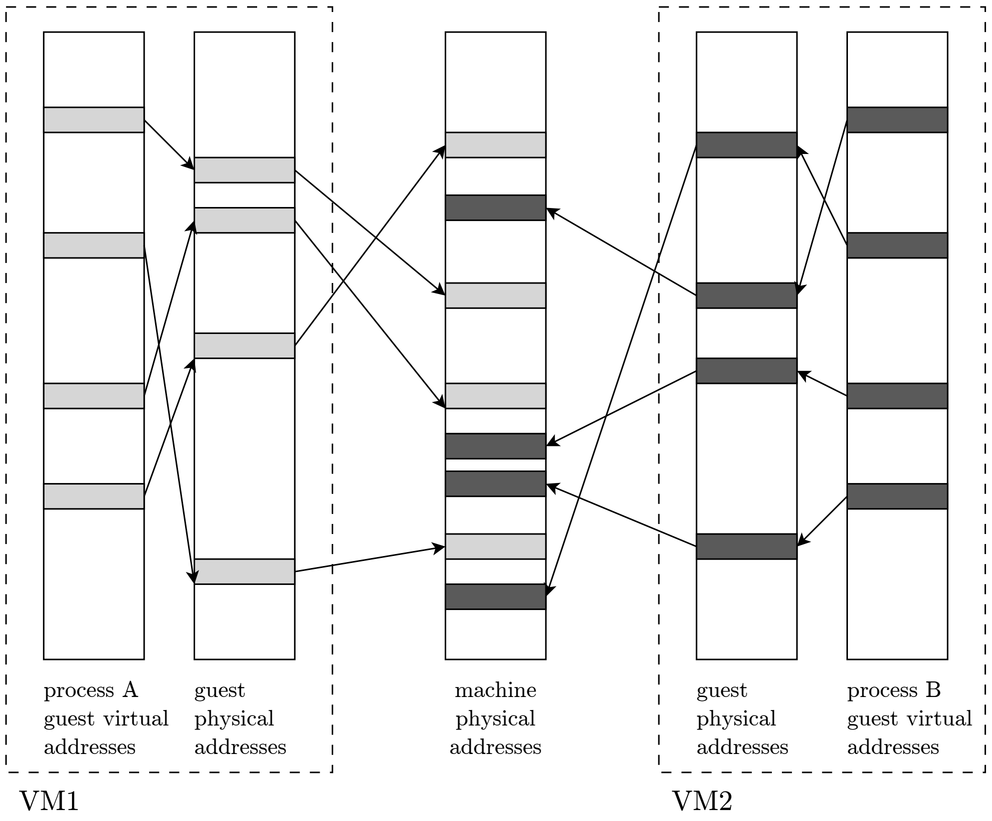

Figure 2.7: Memory virtualization.

2.2.3 I/O devices

Beyond the CPU and memory, a virtual machine needs to access I/O devices, such as Network Interface Cards (NICs) and Graphics Processing Units (GPUs). When virtualizing I/O devices, the hypervisor needs to route I/O requests of guests between the virtual device and the physical device.

With the *emulation* solution, the hypervisor has a software implementation of the features of standard devices, over any physical device. All the commands of the real-world device are replicated in software, and have the same behavior as its hardware counterpart. The common approach is to emulate a well-known real-world device. This way, the guest operating system can run an unmodified device driver. This approach has acceptable performance for basic and low-bandwidth devices.

A *split-driver* (also called *paravirtualized driver*) solution lets a privileged domain handle hardware management. A frontend driver runs in the guest virtual machine and forwards calls to the backend driver in the host. The backend driver then takes care of sharing resources among virtual machines. This solution requires special drivers for the guest virtual machine. However, it has the advantage of multiplexing several guests on the same hardware device, and improves performance compared to emulation.

With the *direct device assignment* solution, the guest virtual machine has direct control of the device. This is essentially achieved by mapping device memory into the virtual machine memory space. This solution requires dedicated hardware to limit the memory accesses of the device, and to re-route interrupts. It also forbids multiplexing, allowing a single guest on a given device. However, it does not require any driver change to the guest, and provides near native performance. It is thus an interesting solution for devices that have complex drivers, and are performance-oriented, such as GPUs.

Single Root I/O Virtualization (SR-IOV) capable devices can expose themselves to the operating system as several devices. With this solution, the hardware device itself can be composed of several independent functions (multiple devices) or multiplex the resources in hardware.

2.2.4 Attack surface of the virtualization layer

The virtualization layer itself is a target, given its privileged access to hardware. The virtualization layer is composed of the hypervisor, the host operating system, as well as an emulator. This layer is quite complex and forms a large trusted computing base. The attacker searches to attack another guest by escaping the guest environment, targeting the hypervisor and performing his attack through privileged access via the host.

2.2.4.1 Hypervisor

Hypervisor exits constitute a first attack surface. They are performed when the guest is interrupted so that the hypervisor handles an event. As Szefer et al. [SKLR11] noted, a guest performs a lot of exits: $\sim$600 per second when idle, and more than 9 million when the guest starts. Each exit is a possibility for exploiting a bug in the hypervisor code. Typical proposed mitigations consist in reducing the trusted computing base. Seshadri et al. [SLQP07] proposed SecVisor, a hypervisor with a tiny code base. However, the lack of functionalities do not make it a practical alternative for cloud providers. Szefer et al. [SKLR11] proposed to remove the hypervisor attack surface altogether, by enabling guests to run natively on hardware, and eliminate all interactions between the hypervisor and guests.

2.2.4.2 Device emulator

Virtual hardware, e.g., virtual network devices, are handled by an emulator such as QEMU. However, the code base of QEMU is hard to maintain: it includes many legacy devices, and low-level code. Vulnerabilities are thus regularly found. As an example, in 2015 Jason Geffner found a vulnerability in the virtual floppy disk controller [CVE15], making possible a privilege escalation regardless of the presence and configuration of the virtual floppy disk. The OpenStack Foundation [Ope13] recommended removing unused components from the system in order to minimize the QEMU code base as a practical defense.

2.2.4.3 Direct device assignment

Physical hardware accessed by direct device assignment also introduces security issues. Certain hardware devices use Direct Memory Access (DMA) to access memory in the operating system in native environments. Arbitrary memory accesses are an issue in virtualized environments because it allows the guest to map its device memory into the memory of the host or into that of other guests. Pék et al. [PLS$^+$14] reviewed these attacks and their feasibility. A hardware I/O Memory Management Unit (IOMMU), such as Intel's VT-d, thwarts DMA attacks by preventing devices from accessing arbitrary parts of the physical memory. However, Wojtczuk and Rutkowska showed that the IOMMU is not exempt from vulnerabilities by building software attacks that bypass it [WR11]. Willmann et al. [WRC08] reviewed different protection strategies for devices in direct device assignment. They argue that software-only strategies can outperform IOMMU-based strategies, with less overhead.

2.3 Covert and side channels on shared resources

We now consider another type of attacks, applicable more broadly to multi-tenant environments, for which virtualization is a natural use case. In the case of virtualization, these attacks do not rely on flaws in the virtualization layer such as described in Section 2.2.4. Instead of directly compromising the other guest, the attacker is a non-privileged process that uses shared hardware as a medium to leak information. These attacks fall into the category of *covert channels* and *side channels*. Covert channels involve the cooperation of two attacker processes to actively exchange information. Side channels imply passive observation of a victim process by an attacker process, usually to extract a secret like a cryptographic key.

2.3.1 Achieving and detecting co-residency

To perform covert or side channels on shared hardware, the first step for an attacker is to be *co-resident* with his victim, *i.e.*, to share a physical machine. In a native environment, the attacker has to run a program on the same operating system as its victim. In a virtualized environment, the attacker has to run a virtual machine alongside the virtual machine of its victim, on the same physical machine. We now review methods for an attacker to achieve and detect co-residency on a virtualized environment.

Ristenpart et al. [RTSS09] presented some heuristics to achieve co-residency on Amazon EC2 instance placement. They started by mapping the IP ranges of EC2 service, that correspond to different instance types and availability zones. The authors also showed that a brute-force strategy already achieves a reasonable success rate, for a large set of victim instances. A more elaborate strategy abuses the placement algorithm, that tends to co-locate virtual machines launched in a short temporal interval. The attacker can also abuse the auto-scaling system that automatically creates new instances when demand increases. This forces the victim to launch a new instance, from which point the attacker can himself launch new instances until one is co-resident with the victim. Varadarajan et al. [VZRS15] reevaluated the co-residency vulnerability in three major cloud providers, after the adoption of Virtual Private Cloud (VPC). VPC logically isolates networks, but it does not gives physical isolation, *i.e.*, virtual machines from different VPCs can share the same physical machine. Varadarajan et al. found that VPC makes prior attacks ineffective. However, Xu et al. [XWW15] demonstrated a new approach to attack instances that are in a VPC. They exploit latency variations in routing between instances that are behind a VPC and instances that are not. This approach comes at high cost, as an attacker needs to launch more than 1000 instances to achieve co-residency in a VPC. However, it shows that virtual network isolation does not completely solve the issue, thus attacks on co-resident virtual machines are still a real threat.

The next step is then to check co-residency. In [RTSS09], the authors also proposed a load-based co-residency test based on requests of a client that performs HTTP requests. With their tool HomeAlone, Zhang et al. [ZJOR11] monitored L2 cache usage to detect co-residency: they used a classical side channel as a defensive tool. A virtual machine that wants to attack a co-resident victim running HomeAlone has to lower its fingerprint, and consequently cannot use the cache as a side channel. Bates et al. [BMPP12] leveraged the sharing of the network interface to detect a co-resident virtual machine via traffic analysis techniques. The attacker needs a flooder virtual machine and a client that makes requests to a victim server. If the flooder is co-resident with the victim server, the flooder can induce a watermark, *i.e.*, a specific pattern, in the traffic from the victim server to the client.

2.3.2 Attack surface of an x86 processor

2.3.2.1 Data and instruction cache

Side and covert channels on both data and instruction caches are based on differences in timing between memory accesses that are in cache, and ones that are served from the main memory.

Data cache attacks exploit secret-dependent data access patterns. They happen when the implementation of cryptographic algorithms relies on lookups that depend on the secret key – e.g., the substitution boxes in block ciphers such as AES, and the multipliers table in public-key algorithms such as RSA. There is a large body of literature on the subject of data cache attacks. These attacks are described in more detail in Section 2.4.

Instruction cache attacks exploit secret-dependent execution paths instead. They work similarly as data cache attacks. Block ciphers like AES are not vulnerable to instruction cache attacks since they have a fixed instruction flow. Aciiçmez [Aci07] used the instruction cache to attack RSA implementations that use Square and Multiply exponentiation algorithm. The sequence of square and multiply operations are key-dependent. A spy process executes dummy instructions to fill the cache before the execution of the cipher process. If the cipher process executes square (resp. multiply) instructions, it evicts the dummy instructions of the attacker, which will result in a cache miss when the attacker resumes the execution of his dummy instructions. The attacker thus reveals the execution flow of the cipher by measuring the execution time of his own instructions. Aciiçmez et al. [ABG10] later improved the method using supervised learning. Zhang et al. [ZJRR12] demonstrated the attack across different virtual machines on Xen hypervisor to extract ElGamal decryption keys.

Microarchitectural countermeasures for securing the data cache – such as randomized mappings – were proposed to be adapted to secure the instruction cache [KASZ13, LWL15]. These studies show that these countermeasures incur a low performance overhead, compared to software countermeasures. They however come at the cost of extra hardware and complexity, and are less flexible than software countermeasures.

2.3.2.2 Branch prediction unit

Aciiçmez et al. [ASK07b] presented an attack against the Branch Target Buffer. The attack leverages the differences in execution time against RSA implementations to reveal their execution flow. A spy process executes a number of branches to fill the Branch Target Buffer. When the cipher executes its branches, the prediction will be *not taken*. If the branch is indeed not taken, the prediction is correct and the Branch Target Buffer is not updated. If the branch is taken, this is a misprediction and the Branch Target Buffer is updated. This evicts one of the spy branches, which results in a misprediction for the spy process and a longer execution of one of its branches. The attacks need a statistical analysis over many measurements to be able to accurately differentiate between the square and the multiply operations. Aciiçmez et al. improved the branch prediction attack in [AKS07, AGS07] to recover almost all the secret key bits in a single execution. All these attacks are limited to be performed by the attacker on the same core as the victim, as the branch prediction unit is not shared across cores.

As a countermeasure, Kong et al. [KASZ13] proposed to change the update policy for the Branch Target Buffer, as well as to add dummy conditional branches, so that there is the same Branch Target Buffer mapping for conditional branches whether the branches are taken or not taken.

2.3.2.3 Arithmetic logic unit

Some functional units are shared between threads on the same core, to allow hyper-threading at a moderate cost in terms of die surface. In particular, Intel Pentium 4 shares a long integer multiplier unit between threads of the same core. While at a given time only one thread can use this unit, hyper-threading interleaves the execution of two independent long integer multiplication threads.

Aciiçmez et al. [AS07] attacked the Square-and-Multiply Exponentiation Algorithm for RSA. The spy process continuously executes multiply operations, and measures the execution time of their own process. The execution is longer if the victim is also performing a multiply operation as well, thus leaking information on the key bits.

2.3.3 Attack surface of the memory and memory bus

2.3.3.1 Memory deduplication

Page-level deduplication is used by hypervisors and operating systems as a means to avoid storing multiple copies of redundant data. In the Linux kernel, the deduplication mechanism is called Kernel Same-Page Merging (KSM), and it is used by KVM. When KSM is active, a thread regularly scans all pages to find pages that are identical. Identical pages are then merged to a single read-only page. A write to this page triggers a copy.

This copy-on-write mechanism is used maliciously for memory disclosure attacks, as it induces a higher latency compared to a regular write access. Suzaki et al. [SIYA11] used it to infer the applications that are running on other guests, whereas Owens et al. [OW11] inferred the operating system of other guests. Xiao et al. [XXHW13] built a covert channel that can attain above 90bps, and 40bps on a system under memory pressure. The authors also showed that memory deduplication can be used to detect virtualization – VMWare ESX server, Xen, and Linux KVM – independently of any instruction or guest OS. Barresi et al. [BRPG15] exploited memory deduplication to defeat Address Space Layout Randomization (ASLR) in virtualized environments, leaking the address layout of co-located virtual machines.

Due to these security concerns, some public cloud providers have now disabled memory deduplication. However, Gruss et al. [GBM15] showed that these attacks can also be ported to JavaScript, enabling an attacker to mount an attack remotely through a website. This places the threat not only on public cloud environment, but on any system that has enabled memory deduplication.

2.3.3.2 Memory bus

Wu et al. [WXW12] proposed a bus-contention based covert channel, that uses atomic memory operations that lock the shared memory bus. Atomic memory operations are guaranteed to perform uninterrupted. On early processor generations, it is implemented by locking the bus, which results in system-wide contention. Next generations improve the implementation by locking the cache line and not the bus, when possible. The bus is still locked when an atomic memory operation is performed on an unaligned address that spans two cache lines. These accesses are quite exotic because modern compilers automatically generate aligned accesses, but they are easy to generate manually.

The Nehalem microarchitecture introduces Non-Uniform Memory Access (NUMA) and removes the shared memory bus. However, exotic memory accesses emulate bus lock to ensure the atomicity of the operation.

The bus contention is visible system-wide, which creates conditions for a covert channel, exploitable even across processors contrary to, e.g., cache-based covert channels. The authors obtained a raw bandwidth of 38kbps between two virtual machines, and about 746bps with a protocol that handles error correction. They further evaluated the covert channel on EC2 virtual machines, obtaining more than 100bps when the sender and receiver are throttled. This covert channel is not affected by non-participating workload that stresses the cache, but as expected it is still affected by memory intensive workloads. Saltaformaggio et al. [SXZ13] designed a hypervisor-based solution to prevent this attack. It hooks page faults, and replaces the atomic instructions – which are responsible for the covert channels – with a trap to the hypervisor.

2.4 The case of the data cache of x86 processors

We now cover attacks on data caches – simply called "cache attacks" in the remainder – in more detail. There are three categories of cache attacks: *time-driven*, *trace-driven*, and *access-driven*. The differences between them are the abilities of the attacker and the number of measurements needed. They are mainly focused on cryptographic side-channel attacks, but some are adapted to build covert channels, or to leak other kind of information.

2.4.1 Time-driven attacks

In a time-driven attack, the attacker only observes the overall execution time of the cipher. He then deduces the total number of cache hits and cache misses from these aggregate measurements, via statistical analysis on many executions. These attacks are the least restrictive, because they do not require strong assumptions on the attacker, like the ability to watch the outcome of memory accesses or special equipment.

Such attacks have been conducted by Tsunoo et al. [TSS03] against DES. Bernstein et al. [Ber05], Neve et al. [NSW06], Bonneau and Mironov [BM06], and Aciiçmez et al. [ASK07a] used it against AES. Weiss et al. [WHS12] and Irazoqui et al. [IIES14a] revisited Bernstein's attack against AES in a virtualized environment.

2.4.2 Trace-driven attacks

In a trace-driven attack, the attacker observes the outcome of memory accesses in terms of cache hits and misses, *i.e.*, the trace of the execution. The attacker makes several measurements by changing the plaintext, that will lead to different access patterns, thus different traces. The attacker then infers the key from these traces. This type of attacks leaks more information than a time-driven attack to an attacker. However, it also supposes a more powerful attacker, who has physical access to the device, complex equipment and signal analysis techniques.

These attacks have been performed against AES by Bertoni et al. [BZB$^+$05], Lauradoux [Lau05] and Aciiçmez et al. [AK06]. The traces are captured using power analysis, making these attacks infeasible remotely.

2.4.3 Access-driven attacks

In an access-driven attack, the attacker determines the cache sets that the cipher process modifies by monitoring his own activity and not his victim's. Access-driven attacks leak more information than time-driven attacks while requiring less abilities from the attacker than trace-driven attacks, e.g., no physical access.

This category of attack is further divided into two types:

1. Prime+Probe, where the attacker fills cache sets,
2. Flush+Reload and Evict+Time, where the attacker evicts cache sets.

Table 2.2 gives an overview of the different access-driven attacks.

In a Prime+Probe attack, the attacker repeatedly accesses values in a table to fill the cache. The victim that runs the cipher eventually evicts entries in the cache. When the attacker regains control, he reads the memory lines to check which ones were evicted from the cache. This attack has been performed against RSA [Per05], AES [OST06, NS06, TOS10, IES15b], and El-Gamal [LYG$^+$15].

In a Flush+Reload attack, the attacker evicts some memory lines, waits while the victim performs encryption, and reloads the lines to check which one were reloaded by the victim. The lines are shared between the attacker and the victim, and evicted using the `clflush` instruction. Due to their ability to target a single line, these attacks are precise and very powerful. Such attacks have been performed against RSA [YF14], AES [GBK11, IIES14b] and ECDSA [BvdPSY14]. Irazoqui et al. [IES15a] used Flush+Reload to revive an attack on CBC encryption that was patched some years before. In an Evict+time

Table 2.2: Overview of access-driven cache attacks

Year	**Article**	**Cipher**	**Method**	**Cores**	**Scope**	**Memory**	**Platform**
2005	[Per05]	RSA	Prime+Probe	single	native	non-shared	Pentium 4
2006	[OST06]	AES	Prime+Probe	single	native	non-shared	Pentium 4E/Athlon 64
	[OST06]	AES	Evict+Time	single	native	non-shared	Pentium 4E/Athlon 64
	[NS06]	AES	Prime+Probe	single	native	non-shared	–
2010	[TOS10]	AES	Prime+Probe	single	native	non-shared	Pentium 4E
2011	[GBK11]	AES	Flush+Reload	single	native	non-shared	Pentium M
2014	[YF14]	RSA	Flush+Reload	multi	virt	shared	Core i5-3470
	[BvdPSY14]	ECDSA	Flush+Reload	multi	virt	shared	–
	[IIES14b]	AES	Flush+Reload	multi	virt	shared	Core i5-3320M
2015	[IES15a]	CBC enc.	Flush+Reload	multi	virt	shared	Core i5-650
	[IES15b]	AES	Prime+Probe	multi	virt	non-shared	Core i5-650
	[LYG$^+$15]	El-Gamal	Prime+Probe	multi	virt	non-shared	Core i5-3470/Xeon E5-2690

attack, the attacker first performs an encryption to measure the execution time. Similarly to Flush+Reload, in the second step the attacker evicts a cache set, and performs the encryption again to compare the execution time with the first one. If the time increases, the attacker knows that the victim program accesses this set. These attacks are akin to time-driven attacks, as they need numerous executions. Such attacks have been performed against AES [OST06].

The differences between Prime+Probe and Flush+Reload attacks lies in the requirements and in the granularity. Flush+Reload attacks require shared memory between the spy and the victim. In this setup, the system loads data (e.g., libraries) in the main memory only once. Data is thus also shared in physically indexed caches, where the spy process can evict it with the `clflush` instruction. However, as Flush+Reload attacks operate on a single cache line, they are more fine-grained than Prime+Probe attacks that operate on sets.

2.4.4 Beyond cryptographic side channels

Most of the aforementioned attacks are side-channel attacks on cryptographic algorithms. However, other type of attacks can be derived from the same techniques.

Covert channels exploit the same techniques as access-based cryptographic side channels, using a variant of the Prime+Probe technique. Ristenpart et al. [RTSS09] constructed a covert channel on the L1 cache, restricting the two programs to be on the same core. They obtained a bitrate of 0.2bps, running cross-VM on Amazon EC2 environment. Xu et al. [XBJ$^+$11] studied the feasibility and quantified cache covert channels, from the theoretical model to the real life implementation. They showed that these covert channels have a bitrate between 85bps and 215bps in laboratory environment, but come to less than a bit per second in Amazon EC2 environment. Indeed, the covert channel is not cross-core, thus the sender and receiver need to share the same core, which does not happen frequently due to the hypervisor scheduling. The authors concluded that covert channels are thus worrying but still limited to the extraction of small secrets. Wu et al. [WXW14] designed a cross-core cache covert channel, obtaining a raw bitrate of 190kbps in a native environment with a Nehalem processor.

Similarly, attacks that target sensitive data other than cryptographic keys leverage the same techniques as access-based side channels. Hund et al. [HWH13] used cache attacks to defeat kernel-space ASLR. Zhang et al. [ZJRR14] used Flush+Reload technique on Platform-as-a-Service (PaaS) clouds to extract data such as the number of items in a shopping cart from applications co-located with a malicious virtual machine. Irazoqui et al. [IIES15] exploited the Flush+Reload technique to detect and distinguish between various popular cryptographic libraries. Oren et al. [OKSK15] showed that cache attacks can be exploited from a web page in JavaScript, using the Prime+Probe technique to, e.g., spy on user mouse activity.

Finally, the exploitation of these attacks is not an easy task: monitoring leaking addresses requires a good understanding of the software that is attacked, and its source code can be very large e.g., including many libraries, or not available *i.e.*, only a binary is accessible. Gruss et al. [GSM15] presented a generic attack method to overcome these difficulties, using the Flush+Reload technique. The genericity of the approach also enables developers to monitor their own piece of software to find if it is susceptible to cache attacks.

2.4.5 Evolutions of cache attacks

Cache attacks changed and developed along three axes. First, they evolved with *hardware*, from single-core to multi-core CPUs. Second, they evolved with *execution environments*: with the rise of the cloud computing, attacks that first targeted native environments migrated to virtualized environments. Third, attacks naturally evolved with the *countermeasures* in production: the most recent attacks do not use any shared memory.

2.4.5.1 From single-core to muti-core CPUs

Cache attacks evolved with hardware. The first attacks targeted current single-core processors, e.g., Intel Pentium 4 [Per05]. They either relied on Simultaneous MultiThreading (SMT) [Per05, OST06, TOS10] that makes one physical core working as two logical cores, or they exploited the OS scheduling when SMT is not available or disabled for security reasons [NS06].

Multi-core began to be the norm in the following generations of CPUs, beginning with the Core microarchitecture in 2006. Consequently, only exploiting one core drastically reduced the performance of attacks. Indeed, the attacker and the victim are no longer always running on the same core, due to the OS scheduling [XBJ+11]. Due to this reason, and the fact that processors became more and more complex, Mowery et al. [MKS12] pinpointed that side-channel attacks were increasingly more difficult. However, multi-core processors often have a last-level cache that is shared between cores. Yarom and Falkner [YF14] exploited this shared last-level cache and shared memory to construct the first cross-core cache side channel, with Flush+Reload.

Cross-core attacks that do not use shared memory need to be able to precisely locate cache lines in the last-level cache. However, the design of the last-level cache also changed with generations of microarchitecture. Sandy Bridge introduced the complex addressing function that maps addresses to last-level cache slices, whereas Nehalem uses only one or two bits of the physical address. This function is not documented, it is thus not possible to locate a cache line in the last-level cache just by knowing its physical address. Hund et al. [HWH13] reverse-engineered the function for a Sandy Bridge processor, but no study had been made to generalize this finding. Irazoqui et al. [IES15b] targeted processors pre-Sandy Bridge. Concurrently, Liu et al. [LYG$^+$15] bypassed the complex addressing function, and achieved the evcition of particular cache lines by building sets of addresses, called *eviction sets*, with a timing attack. Their attack is thus able to target more recent processors.

2.4.5.2 From native to virtualized environments

Research on cache side channels started with theoretical attacks in 1992, with Hu [Hu92], followed by Kocher et al. [Koc96] and Page [Pag02]. The first practical attack was demonstrated by Tsunoo et al. [TSS03], on a native environment. Research on cache side channels got a new impetus with the rise of cloud computing, in particular for time-driven and access-driven attacks that do not rely on physical access. In 2009, Ristenpart et al. [RTSS09] showed that the CPU cache was a means of information leakage between two virtual machines on Amazon EC2 cloud platform. Zhang et al. [ZJRR12] were the first to demonstrate the use of side channels to extract cryptographic keys across virtual machines using the instruction cache, in a laboratory environment. Since then, the following articles mostly consider cross-VM leakage [YF14, IIES14b, IES15a, IES15b, LYG$^+$15].

2.4.5.3 From shared to non-shared memory

Flush+Reload attacks operate at the granularity of a single cache line, they are thus very powerful. However, they require shared memory. As some public cloud providers disabled shared memory for security reasons, the most recent attacks use the Prime+Probe technique. At the beginning of 2015 and concurrently to this thesis, Irazoqui et al. [IES15b] and Liu et al. [LYG$^+$15] demonstrated that it is possible to target a precise set without any shared memory on the physically indexed last-level cache. Both articles exploit 2MB pages that are contiguous in physical memory.

2.4.6 Timing measurements

Establishing cache covert and side channels demands fine-grained and accurate timing measurements. Processors provide a timestamp counter for the number of cycles since reset. This counter can be accessed by the `rdtsc` and `rdtscp` instructions. However, because of the out-of-order execution, the actual execution may not respect the sequence order of instructions as written in the executable. In particular, a reordering of the `rdtsc` instruction can lead to a measurement of more, or less, than the wanted sequence. Reordering can be prevented by using serializing instructions, such as `cpuid`. Intel gives recommendations for fine-grained timing analysis in [Int10]. Appendix B gives a listing of the code we used to perform timing measurements in the remainder of this thesis.

2.4.7 Countermeasures

Countermeasures against cache attacks can be envisioned at three levels: directly at the architecture or microarchitecture level, at the operating system or hypervisor level, and finally, at the application level.

2.4.7.1 Architecture or microarchitecture level

Instruction set Some changes to the instruction set itself can mitigate or completely remove side channels. For example, on ARM architecture, flushing a line is a privileged operation – contrary to the x86 `clflush` instruction –, thus removing the Flush+Reload attack vector for this architecture. Additionally, Intel introduced an extension to its x86 instruction set, called AES-NI [Int08]. This added new instructions to enable data encryption and decryption with the AES algorithm. These instructions have a fixed time – thus countering time-driven attacks – and remove memory accesses on key-dependent data – thus countering trace-driven and access-driven attacks. This countermeasure however implies a change in existing programs to use these instructions, and only works on supported hardware.

Secure cache designs Two general solutions are proposed regarding new secure cache designs: either *removing* cache interferences, or *randomizing* them such that information about cache timings is useless to the attacker.

To remove cache interferences, Page [Pag05] proposed partitioning the cache architecture, as well as introducing changes to the instruction set. This approach however is quite heavy and causes performance degradations. Wang and Lee [WL06, WL07] proposed the Partition Locked Cache (PLcache). It avoids cache interference by dynamically locking cache lines of sensitive programs, thus preventing other processes from evicting them. After analyzing the security of PLcache [KASZ08], Kong et al. [KASZ09, KASZ13] proposed an improvement over its logic: preloading all critical data in cache before the beginning of the cryptographic operations. All critical data will thus be locked in cache. Domnister et al. [DJL+11] proposed to modify the replacement policy in the cache controller. It statically reserves one or several ways for each hardware thread, without the need for the instruction set to change.

To randomize cache interferences, Wang and Lee [WL06, WL07] proposed the Random Permutation Cache (RPcache). It creates permutations tables so that the memory-to-cache-sets mapping are not the same for sensitive programs as for others. After analyzing the security of RPcache [KASZ08], Kong et al. [KASZ09, KASZ13] proposed two improvements over it. The first is the use of *informing loads* to secure RPcache. Informing loads are special instructions that inform the software when a load misses in the cache. An exception can then be raised, and the exception handler can then load critical data such that future loads will hit the cache, eliminating time variations. The second is to use informing loads to change the permutations of RPcache when critical data misses the cache. Wang and Lee [WL08] introduced a third secure cache design, called Newcache, that also randomizes the interferences. It adds a level of indirection for memory-to-cache mapping, as well as a modified random replacement algorithm. Liu and Lee [LL13] investigated the security of Newcache, and proposed a modification to the replacement algorithm to counter specifically crafted attacks. Liu and Lee [LL14] proposed to change the filling policy of the cache, to de-correlate it from the demand. It means that instead of filling the cache for each miss, data populates the cache after being served to the processor, randomly within a configurable time window.

Prefetcher Fuchs and Lee [FL15] suggested to use disruptive prefetching in order to mitigate cache side channels. Prefetchers are traditionally studied regarding performance. In terms of security, the key idea is that the prefetcher adds noise to the original memory access sequence. By altering the prefetching policy, the cache behavior becomes less predictable. Their new prefetching policy prevents attacks on the L1 data cache, and has been tested against a Prime+Probe attack.

2.4.7.2 Operating system or hypervisor level

Isolation A first countermeasure at the system level – operating system or hypervisor – is to isolate the different virtual machine processes, so that they do not share the resources that are the cause of information leakage. The most obvious solution is to only allow one virtual machine per physical machine. However, this removes the main benefit of virtualization in terms of performance and thus raises the cost of cloud computing. A more fine-grained solution is to physically isolate only previously annotated functions [BJB15]. Similarly to new cache designs, page coloring provides cache isolation, but operating at the software level [RNSE09, SSCZ11, KPMR12]. Other papers have a more relaxed isolation approach in software. Zhang et al. [ZR13] proposed Düppel, that repeatedly cleans caches that are time-shared, e.g., the L1 cache. Varadarajan et al. [VRS14] investigate the role of the scheduler to limit the frequency of cross-VM interactions. Although it does not completely eliminate the possibility of such interactions, side channels require frequent measurements to be accurate. Changing the scheduler to limit the frequency of preemptions is thus a practical way of defeating side channels on private caches in virtualized environments. This countermeasure however needs to be evaluated against the newer cache attacks on the shared last-level cache.

Noise in timers Another solution is to use lower-resolution timers or remove them altogether [VDS11, MDS12]. Indeed, one condition for side channels is the ability for the attacker to perform fine-grained measurements to distinguish between, e.g., a cache hit and a cache miss. However, this solution does not account for legitimate uses of fine-grained timers.

Normalized timings Deterministic execution in cloud architecture is an alternative solution to remove timing channels [AHFG10, LGR13]. Similar to this idea, Braun et al. [BJB15] proposed modifying the OS to offer protection to sensitive functions, previously annotated. Here the key idea is that the execution time is not entirely deterministic: only the key-dependent computations are time-padded, not external timing differences (e.g., OS scheduling, CPU frequency scaling).

2.4.7.3 Application level

Finally, countermeasures can be built directly at the application level. They are specific to each application, and require either changes in the compilation flow, or in the algorithms themselves. Since they focus on specific applications, they are not able to prevent all cache attacks, e.g., covert channels.

Compiler-based mitigations Compiler-based mitigations have the advantage to automate parts of the changes needed to prevent side channels. Coppens et al. [CVDD09] proposed to use automated compiler techniques to remove key-dependent control flow and key-dependent timings in cryptographic software. The changes are made in the backend compiler, leveraging x86 conditional move instructions to eliminate branches. Subsequently, Cleemput et al. [CCD12] investigated variable latency instructions. They evaluated changes to the backend compiler that either compensate this variable latency, or force the use of fixed latency operations. They conclude that these transformations incur a too high overhead when strong protection is required, thus making this solution not practical. Crane et al. [CHB+15] explored software diversity. Their key idea is to create several clones of sensitive program fragments, that are functionally equivalent but differ in runtime characteristic. Then at runtime, the program dynamically and randomly chooses which control path to take.

Manual changes in applications Manual changes in applications are tedious since they require finding the source of the leak and to patch it manually, and they are dependent on the algorithm and implementation. For example, Brickell et al. [BGNS06] focused on AES and proposed compressed and randomized tables, as well as pre-loading cache lines. However, Blömer and Krummel [BK07] showed that these countermeasures are sometimes not sufficient. For AES, bitslice implementations avoid using table lookups [RSD06, KÖ8, KS09], without any change in hardware.

They are nonetheless necessary in real-world critical software, since the developers cannot assume changes on the hardware or operating system. For instance, OpenSSL includes mitigations against Percival [Per05] attack[1], against Aciiçmez et al. [AGS07] branch prediction attack[2], and against Yarom et al. [YB14] attack against ECSDA[3]. To that effect, methods that detect side channels in binaries can be used to find the source of information leakage and to close it [DFK+13, GSM15].

[1]See [Ope], "Changes between 0.9.7g and 0.9.7h", 11 October 2005.
[2]See [Ope], "Changes between 0.9.8e and 0.9.8f", 11 October 2007.
[3]See [Ope], "Changes between 1.0.1f and 1.0.1g", 7 April 2014.

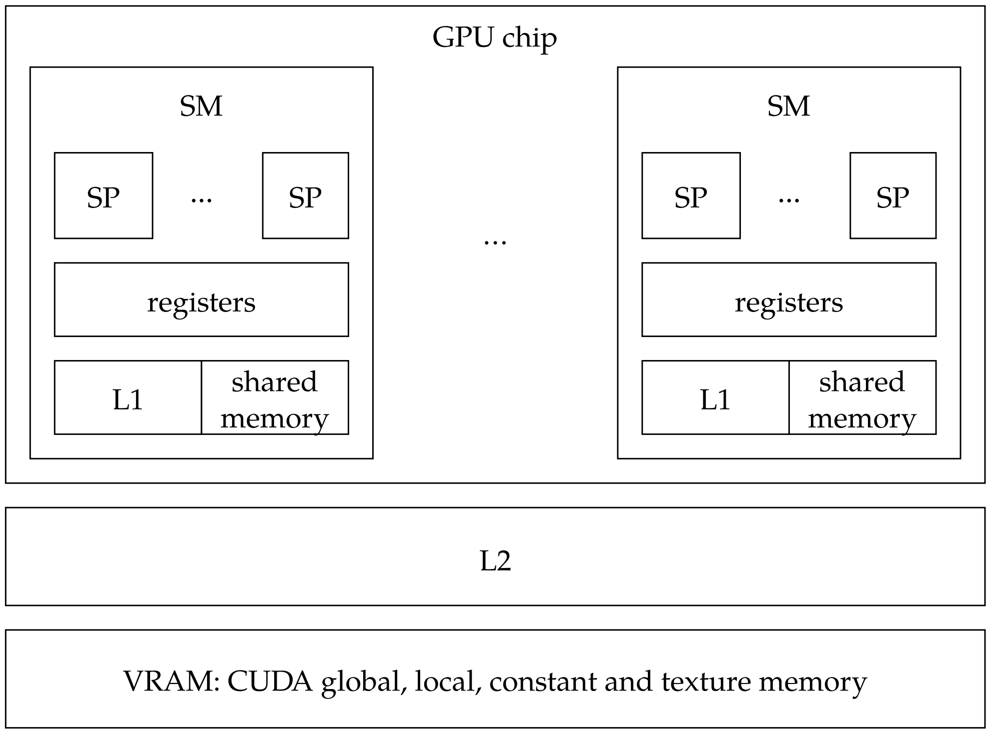

Figure 2.8: Architecture of NVIDIA GPU.

2.5 Information leakage on GPU memory

In this section, we investigate the GPU memory. We focus on NVIDIA GPUs in the remainder of this thesis, considering that they are the most widespread devices used for GPGPU applications. We review their architecture, as well as their usage in security – offensive and defensive.

2.5.1 Architecture

In 2006, NVIDIA launched the Tesla microarchitecture[4] that introduced a general purpose pipeline. Tesla microarchitecture is followed by Fermi, Kepler and the latest to date, Maxwell.

GPUs handle throughput-based workloads that have a large degree of data parallelism. They have hundreds to thousands of cores that can handle

[4]Tesla is used by NVIDIA both as an architecture code name and a product range name [LNOM08]. NVIDIA commercialized the Tesla microarchitecture under the name GeForce 8 Series. When not specified, we refer to the product range name.

hundreds of threads. Data parallelism mitigates the latency caused by the limited memory bandwidth and the deep pipeline. Figure 2.8 illustrates the architecture of a GPU. It is first composed of several streaming multiprocessors (SM), which are in turn composed of streaming processor cores (SP, or CUDA cores). The number of SMs depends on the card, and the number of SP per SM depends on the architecture. The Fermi architecture introduces a memory hierarchy. It offers an off-chip DRAM memory and an off-chip L2 cache shared by all SMs. On-chip, each SM has its own set of registers and its own memory partitioned between a L1 cache and a shared memory accessible by the threads running on the SPs.

2.5.2 Programming model

CUDA is the most used GPGPU platform and programming model for NVIDIA GPUs. CUDA allows developers to write GPGPU-specific C functions called *kernels*. Kernels are executed n times in parallel by n threads. Each SP handles one or more threads. A group of threads is called a block, and each SM handles one or more blocks. A group of blocks is called a grid, and an entire grid is handled by a single GPU. CUDA introduces a set of memory types. Global, texture and constant memory are accessible by all threads of a grid and stored on the GPU DRAM. Local memory is specific to a thread but stored on the GPU DRAM. Shared memory is shared by all threads of a block and stored in shared memory. Finally, registers are specific to a thread and stored on-chip.

CUDA programs either run on top of the closed source NVIDIA CUDA runtime or on top of the open-source Gdev [KMMB12] runtime. The NVIDIA CUDA runtime relies on the closed-source kernel-space NVIDIA driver and a closed-source user-space library. Gdev supports the open source *Nouveau* driver [Nou], the PSCNV driver [Pat] and the NVIDIA driver. Both closed-source and open-source solutions support the same APIs: CUDA programs can be written using the runtime API, or the driver API for low-level interaction with the hardware [NVI12].

2.5.3 Offensive usage of GPUs

2.5.3.1 The GPU as the subject of attacks

Using the CUDA framework, Di Pietro et al. [DLV13] showed that GPU architectures are vulnerable to information leakage, mainly due to memory isolation issues. The leakage affects the different memory spaces in GPU:

global memory, shared memory, and registers. Di Pietro et al. also showed that current implementations of AES cipher that leverage GPUs allow recovering both plaintext and encryption key in the GPU global memory.

Subsequently to our work, Lee et al. [LKKK14] showed that uninitialized GPU memory can be exploited for, e.g., inferring the browsing history on browsers that use GPU-accelerated rendering.

2.5.3.2 The GPU as a medium for attacks

In addition to being the subject of attacks, some work searched to leverage the GPU as a medium for attacks. A first category of attacks exploits the high degree of parallelism for, e.g., password cracking [MKS10, Spr11]. A second category exploits the particular access of the GPU to the system to evade detection. Vasiliadis et al. [VPI10] implemented GPU-assisted malware with unpacking routines and run-time polymorphism [VPI10]. Other work exploits the DMA capabilities of the GPU. In particular, Ladakis et al. [LKV$^+$13] performed keystroke logging, and Danisevskis et al. [DPS13] constructed a privilege escalation attack in a mobile environment.

2.5.4 Defensive usage of GPUs

The computing power of GPUs is also used for defense. Several publications have used GPUs to implement and accelerate cryptographic algorithms, both for asymmetric [HW09] and symmetric block ciphers [GBM12]. However, as Di Pietro et al. [DLV13] showed, some implementations can leak secret keys due to the poor memory isolation of GPUs. Tackling this issue, Vasiliadis et al. [VPAI14] leveraged specific features of the GPU to secure cryptographic operations such as AES and RSA, in a tool called PixelVault. The secret keys are stored in the GPU registers, making them inaccessible to other processes. All the operations are performed entirely on the device, using a GPU kernel that runs indefinitely.

Another usage for GPUs for defensive usage is security monitoring. Lombardi et al. [LP10] leveraged unused GPUs in cloud environments, by monitoring guest virtual machines on the host side. A similar usage is intrusion detection. Indeed, these systems use pattern matching against a large set of regular expressions, and need a high throughput to monitor networks. GPUs thus outperform CPUs in these setups [VAP$^+$08, VPA$^+$09].

Finally, Bress et al. [BKS13] considered using the GPU memory vulnerabilities to perform forensic investigations. Nevertheless, they noted that calls to the CUDA API cannot be guaranteed to not modify the memory.

2.6 Conclusions

In this chapter, we introduced background on x86 systems, covering in more detail the CPU data cache. We also described the virtualization of such systems. Virtualization is widely used today in cloud computing environments that allow several tenants to share a single physical machine, through virtual machines. We detailed well-known covert and side channels on the processor or the memory, with a focus on attacks on the data cache. These attacks gained a new impetus with the rise of cloud computing, since they can be operated remotely via virtual machines sharing same physical machine. However, they are highly dependent on the microarchitecture of the processor, which has changed every year the past few years, and for which some parts are not documented. Attacks also evolved with new countermeasures in production. Indeed, shared memory is now disabled on some cloud environments, which makes attacks that rely on it impractical in real life virtualized environments.

Another recent evolution in cloud computing environments is the rise in usage of GPUs, either for their computing or graphical powers. There is little research on the security of these devices that have been designed for performance, but a previous article has shown GPUs suffer from security issues involving memory isolation. However, no research has been done on the security of these devices in virtualized environments.

In the rest of this thesis, we investigate cache attacks without any shared memory. We also detail how we reverse-engineered an undocumented part of the cache addressing. Finally, we examine the security of GPU virtualization.

3

Bypassing cache complex addressing: C5 covert channel

Contents

3.1 Introduction

Covert channels are used to exfiltrate sensitive information, and can also be used as a co-residency test [ZJOR11]. There are many challenges for covert channels across virtual machines. First, core migration drastically reduces the bitrate of channels that are not cross-core [XBJ+11]. Second, simultaneous execution of the virtual machines over several cores prevents a strict round-robin scheduling between a sender and a receiver [WXW12]. Third, virtual to physical address translation and functions that map an address to a cache set are not exposed to processes, and thus induce uncertainty over the location of data in the cache. This *addressing uncertainty* (term coined in [WXW12]) prevents a sender and a receiver to agree on a particular location to work on.

Covert channels that do not tackle the *addressing uncertainty* are limited to use private first level caches on modern processors. This dramatically reduces the bitrate in virtualized environments. A way to circumvent this issue is to rely on deduplication offered by the hypervisor or the OS. With deduplication, common pages use the same caches lines. However, deduplication is deactivated by some cloud providers [BRPG15], which makes this attack impractical in some scenarios.

In this chapter, we present a novel cache covert channel, called C5. We differentiate our covert channel with previous work by tackling the issue of the *addressing uncertainty* without relying on any shared memory. Our covert channel works between two virtual machines that run across any cores of the same processor. It leverages the shared and inclusive feature of the last-level cache in modern processors. We obtain high bitrates of 1291bps for a native setup and 751bps for a virtualized setup, arguing that this covert channel is practical.

Section 3.2 details the issue of *addressing uncertainty*. Section 3.3 gives an overview of C5. More details are given on the sender in Section 3.4 and on the receiver in Section 3.5. We evaluate our covert channel in a native and virtualized environment in Section 3.6. We discuss the factors that impact performance in Section 3.7, possible countermeasures in Section 3.8, and differences with the related work in Section 3.9. Finally, Section 3.10 concludes and gives perspective on future work.

3.2 The issue of *addressing uncertainty*

Existing cache covert channels rely on the fact that a sender and a receiver are able to target a specific cache set. However, two conditions individually create uncertainty on the addressing, making it difficult to target a specific set of the last-level cache: memory virtualization and complex addressing.

Processors implement virtual memory using an MMU that maps virtual addresses to physical addresses. With virtual machines, hypervisors introduce an additional layer of translation, as illustrated by Figure 2.7 page 16. The guest virtual pages are translated to the guest physical pages, and further

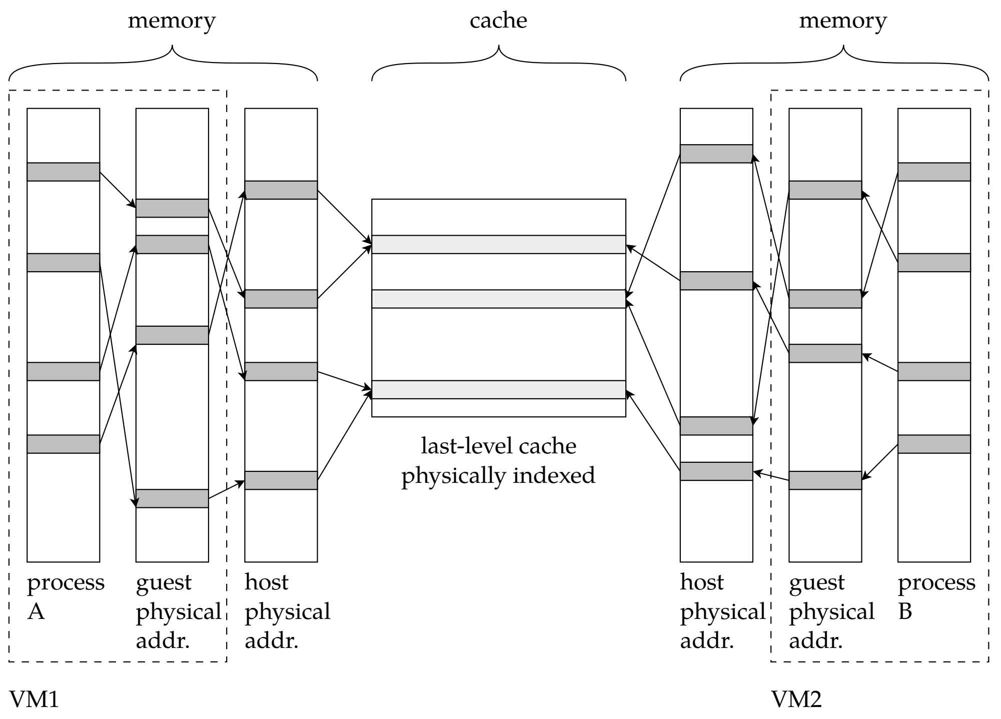

Figure 3.1: Additional layer of indirection added by memory virtualization and its effect on the addressing of the last-level cache sets.

to the actual machine pages. The hypervisor is responsible for mapping the guest physical memory to the actual machine memory. A process knowing a virtual address in its virtual machine has no way of learning the corresponding physical address of the guest, nor the actual machine address. As the last-level cache is physically indexed, this results in the process being unable to predict in which set its access goes (see Figure 3.1).

The complex addressing scheme maps an address to a set with a function that potentially uses all address bits (see Figure 2.5 page 11). As the function is undocumented, a process cannot straightforwardly determine the set in which it is reading or writing. Thus, on processors that use complex addressing, two processes cannot agree on a cache set to communicate, even in a native environment.

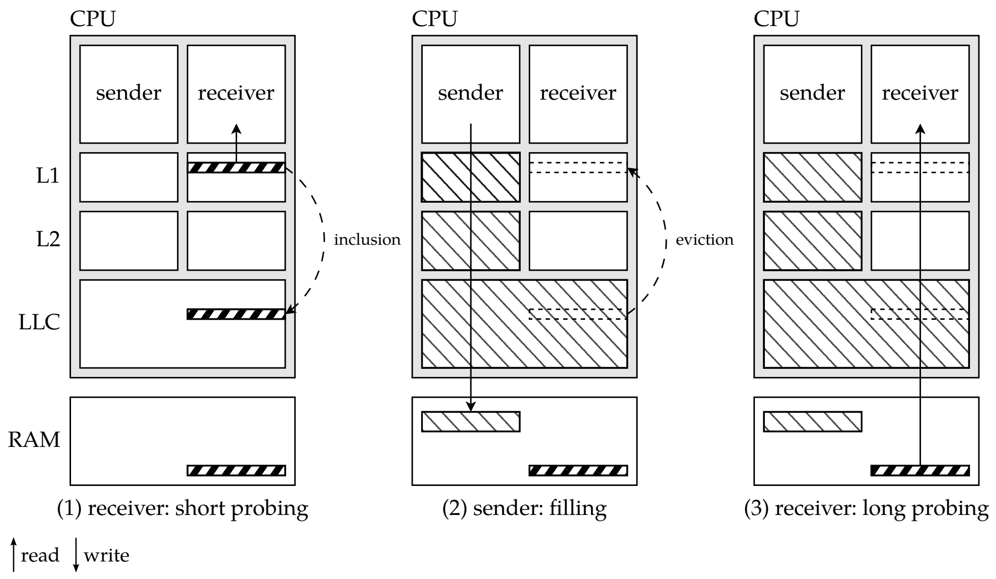

Figure 3.2: Cross-core covert channel illustration of sender and receiver behavior. Step (1): the receiver probes one set repeatedly; the access is fast because the data is in its L1 (and LLC by inclusive feature). Step (2): the sender fills the LLC, thus evicting the set of the receiver from LLC and its private L1 cache. Step (3): the receiver probes the same set; the access is slow because the data must be retrieved from RAM.

3.3 Overview of C5

We now present C5, our cross-core cache covert channel. The sender process sends bits to the receiver by varying the access delays that the receiver observes when accessing a set in its private cache. At a high level view, this covert channel encodes a '0' as a fast access for the receiver and a '1' as a slow access.

Figure 3.2 illustrates our covert channel. It relies on the fact that the last-level cache is shared and inclusive. The strategy is a variant of Prime+Probe. The receiver process repeatedly probes one set. To transmit a '0', the sender stays idle. The receiver observes a fast access, because the data stays in its private L1 cache, see Figure 3.2-1. The data is also present in the last-level cache because of its *inclusive* property. To transmit a '1', the sender process writes data to occupy the whole last-level cache, see Figure 3.2-2. In particular this evicts the set of the receiver from the last-level cache. The data also gets evicted from the private L1 cache of the receiver due to the *inclusive* property. The receiver now observes that the access to its set is slow because the data must be retrieved from RAM, see Figure 3.2-3.

Next we provide a detailed description of the sender and the receiver.

3.4 Sender

To perform a cross-core covert channel, the sender needs a way to interfere with the private cache of the other cores. In our covert channel, the sender leverages the inclusive feature of the last-level cache. As the last-level cache is shared amongst the cores of the same processor, the sender may evict lines that are owned by other processes, and in particular processes running on other cores.

A straightforward idea is that the sender writes in a set, and the receiver probes the same set. However, due to virtualization and complex addressing, the sender and the receiver cannot agree on the cache set to work on. Our technique consists of a scheme where the sender evicts the whole last-level cache, and the receiver probes a single set. This way, the sender is guaranteed to affect the set that the receiver reads, thus resolving *addressing uncertainty*.

In order to evict the whole last-level cache, the sender must evict cache lines and therefore access memory to load new cache lines that will evict the older ones. To do so, the sender just writes data into a buffer. In fact, either writing or reading data would provoke a cache miss. We choose to write because a read miss following a write operation induces a higher penalty for the receiver than a read miss following a read operation. This leads to a stronger signal. We further discuss this choice in Section 3.7.

The eviction of cache lines from the last-level cache leverages the replacement policy. The replacement policy and the associativity influence the buffer size b of the sender. Considering a pure LRU policy, writing n lines in each set is enough to evict all lines of the last-level cache, n being the associativity. Typical replacement policies are pseudo-LRU, variants of LRU and bimodal insertion policy (BIP) where the CPU can switch between the two strategies to achieve optimal cache usage. Pseudo-LRU policies are known to be inefficient for memory intensive workloads of working sets greater than the cache size. The actual replacement policy is a Quad-Age LRU on Ivy Bridge, however, its internals are not documented. We thus determine experimentally the size b of the buffer to which the sender needs to write.

The order of writes into the buffer depends on the cache parameters. Linearly iterating over the buffer would lead to iterate over sets and evict a single

Algorithm 1 Sender: $f(n, o, s, c, w)$

```
message ← {0,1}*
$n$ ← LLC associativity
$o$ ← $\log_2$(line size)
$s$ ← $\log_2$(number of sets in LLC)
$b \leftarrow n \times 2^{o+s} \times c$
buffer[$b$]
for each bit in message do
    wait($w$)
    if bit == 1 then
        for $i = 0$ to number of sets do
            for $j = 0$ to $n \times c$ do
                buffer[$2^o i + 2^{o+s} j$] = constant
            end for
        end for
    end if
end for
```

line of each set before going through the first set again. With too many sets, there is a risk that the receiver probes a set before the sender evicts all lines of this set. In this setting, the probing of the receiver may evict all lines of the sender and the signal may be lost. Ideally, to iterate over the buffer we would take into account the function that maps an address to a set. However, this function is undocumented, thus we assume a direct addressing; other types of iterations are possible. The sender writes with the following memory pattern $2^o i + 2^{o+s} j$ as described in Algorithm 1. 2^s is the number of sets of the last-level cache and 2^o the line size; j and i are line and set indices respectively.

Algorithm 1 summarizes the steps performed by the sender. The parameters are the last-level cache associativity n, the number of sets 2^s, the line size 2^o, and a constant c to adapt the buffer size. To send a '1', the sender evicts the entire last-level cache by writing in each line j ($n \times c$ times) of each set i, with the described memory pattern. To send a '0', the sender does nothing. The sender waits for a determined time w before sending a bit to allow the receiver to distinguish between two consecutive bits.

Algorithm 2 Receiver: $f(n, o, s)$

```
$n \leftarrow$ L1 associativity
$o \leftarrow \log_2$(line size)
$s \leftarrow \log_2$(number of sets in L1)
buffer[$n \times 2^{o+s}$]
**loop**
  read $\leftarrow 0$
  begin measurement
  **for** $i = 0$ to $n$ **do**
    read $+=$ buffer[$2^{o+s}i$]
  **end for**
  end measurement
  record $(localTime, accessDelay)$
**end loop**
```

3.5 Receiver

The receiver repeatedly probes all the lines of the same cache set in its L1 cache. Algorithm 2 summarizes the steps performed by the receiver. The iteration is dependent on the cache parameters. To access each line i (n times) of the same set, the receiver reads a buffer – and measures the time taken – with the following memory pattern: $2^{o+s}i$. The cumulative variable `read` prevents optimizations from the compiler, by introducing a dependency between the consecutive loads so that they happen in sequence and not in parallel. In the actual code, we also unroll the inner `for` loop to reduce unnecessary branches and memory accesses.

The receiver is able to probe a set in its L1 cache because the L1 is virtually indexed, and does not use complex addressing. We do not seek to probe the L2 or L3, because all read and write accesses reach the L1 first and they might evict each other, creating differences in timing that are not caused by the sender.

The receiver probes a single set when the sender writes to the entire cache, thus one iteration of the receiver is faster than one iteration of the sender. The receiver runs continuously and concurrently with the sender, while the sender only sends one bit every w microseconds. As a consequence, the receiver performs several measurements for each bit transmitted by the sender. The receiver could be monitoring a single line from its L1 set, however, this leads to noise if this line is evicted by another program. As the sender is the limiting factor in this covert channel, we choose for the receiver to probe all lines of

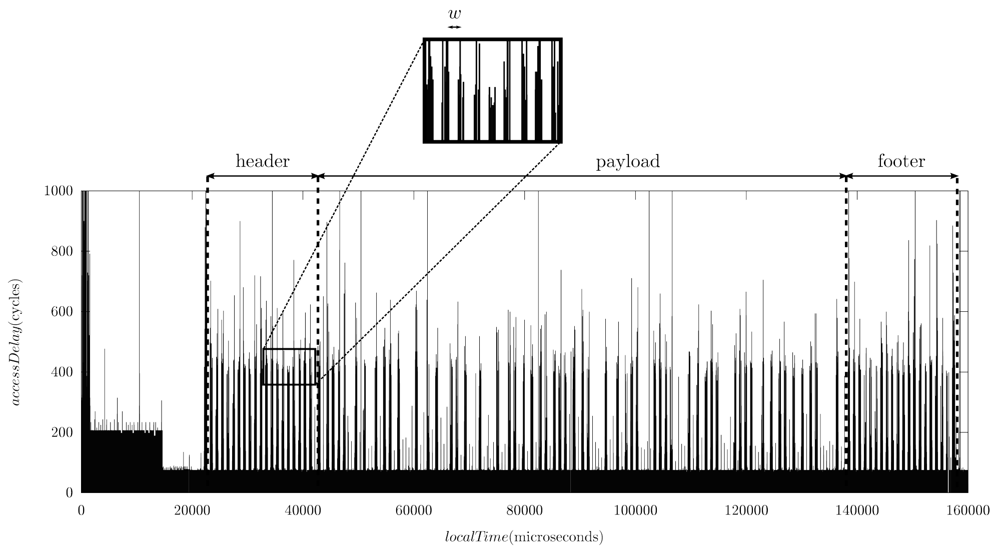

Figure 3.3: Reception of a 128-bit transmission. *Laptop* setup in native environment, with $w = 500$µs, $b = 3$MB.

the set, *i.e.*, to measure the access time of all lines of the set. This smooths the measurements without incurring a performance penalty. Indeed, it reduces the probability of a false positive, *i.e.*, a line has been evicted by another process and is wrongfully counted as a '1'.

Each measurement of the receiver has the form $(localTime, accessDelay)$, where $localTime$ is the time of the end of one measurement according to the local clock of the receiver and $accessDelay$ is the time taken for the receiver to read the set. Figure 3.3 illustrates the measurements performed by the receiver.

Having these measurements, the receiver decodes the transmitted bit-sequence. First, the receiver extracts all '1's. It removes all points that have an $accessDelay$ below (or equal to) typical L2 access time. Then the receiver only keeps the $localTime$ information and applies a clustering algorithm to separate the bits. We choose DBSCAN [EKSX96], a density-based clustering algorithm, over the popular k-means algorithm. A drawback of the k-means algorithm is that it takes the number k of clusters as an input parameter. In our case, it would mean knowing in advance the number of '1's, which is not realistic. The DBSCAN algorithm takes two input parameters, $minPts$ and ϵ:

Table 3.1: Experimental setups, LLC characteristics.

Name	***laptop***	***workstation***
Model	Core i5-3340M	Xeon E5-2609v2
Microarchitecture	Ivy Bridge	Ivy Bridge
Cores	2	4
Size	3MB	10MB
Sets	4096	8192
Associativity	12	20
Complex addressing	yes	yes

1. $minPts$: the minimum number of points in each cluster. If $minPts$ is too low, we could observe false positives, reading a '1' when there is none; if $minPts$ is too high, we could observe false negatives, not reading a '1' when there is one. In practice, we use $minPts$ between 5 and 20.

2. ϵ: if a point belongs to a cluster, every point in its ϵ-neighborhood is also part of the cluster. In practice, we choose ϵ below w.

Once all '1's of the transmitted bit-sequence have been extracted, the receiver reconstructs the remaining '0's. This step is straightforward as the receiver knows the time taken to transmit a '0' which is w.

3.6 Evaluation

In this section, we evaluate our covert channel on native and virtualized setups.

3.6.1 Testbed

Table 3.1 summarizes characteristics of the *laptop* and *workstation* setups. Some parameters of the architecture are constant for the considered processors. The line size in all cache hierarchy is 64 bytes, and the L1 is 8-way associative and has 64 sets. We conduct our experiments in lab-controlled native and virtualized environments.

We adjust two parameters: the size b of the buffer that evicts the last-level cache, and the delay w between the transmission of two consecutive bits. The size b and the delay w impact the bitrate and the error rate of the clustering algorithm, as depicted in Figures 3.4 and 3.5. The precision of the clustering algorithm increases with the size b, however, the bitrate is proportionally reduced. The size b is controlled by the multiplicative parameter c and must be at least the size of the last-level cache. In practice, we used $c = 1$ and $c = 1.5$. The bitrate increases with lower values of w, but the precision of the clustering algorithm decreases.

To evaluate this covert channel, the sender transmits a random 4096-bit message to the receiver. We transmit series of 20 consecutive '1's as a header and a footer framing the payload to be able to extract it automatically. The receiver then reconstructs the message from its measurements. We run 10 experiments for each set of parameters, and calculate the bitrate and the error rate. We derive the error rate from the Levenshtein distance between the sent payload and the received payload. The Levenshtein distance is the minimum number of characters edits and accounts for insertions, deletions and bit flips. We provide evaluation results for each environment: native in Section 3.6.2 and virtualized in Section 3.6.3.

Establishing cache-based channels demands fine-grained and accurate time measurements. However, out-of-order execution creates issues, as explained in Section 2.4.6. We used the code provided in Appendix B to perform our timing measurements.

3.6.2 Native environment

We evaluate C5 in the *laptop* and *workstation* setups, in a native (non-virtualized) environment. We run the sender and the receiver as unprivileged processes, in Ubuntu 14.04. To demonstrate the cross-core property of our covert channel, we pin the sender and the receiver to different cores[1]. Figure 3.3 illustrates a transmission of 128 bits in the *laptop* setup, for $w = 500$µs and $b = 3$MB.

Figure 3.4 presents the results in the *laptop* setup, for two values of b, and three values for waiting time w. For $b = 3$MB (the size of the LLC), we obtain a bitrate between 232bps and 1291bps by varying w. The error rate is comprised between 0.3% (with a standard deviation $\sigma = 3.0 \times 10^{-3}$) and 3.1% ($\sigma = 0.013$). When we increase b to 4.5MB, the bitrate slightly decreases but stays in the same order of magnitude, between 223bps and 1033bps. The error rate decreases between 0.02% ($\sigma = 8.5 \times 10^{-5}$) and 1.6% ($\sigma = 1.1 \times 10^{-4}$). The standard deviation of the error rate also decreases, leading to a more reliable transmission. We conclude that it is sufficient to write n lines per set, but that the transmission is more reliable if we write more than n lines. This is a tradeoff between the bitrate and the error rate.

In the *workstation* setup, we obtain a bitrate of 163bps for $b = 15$MB ($1.5 \times$ LLC), for an error rate of 1.9% ($\sigma = 7.2 \times 10^{-3}$). We observe that the bitrate

[1]Using the `sched_setaffinity(2)` Linux system call.

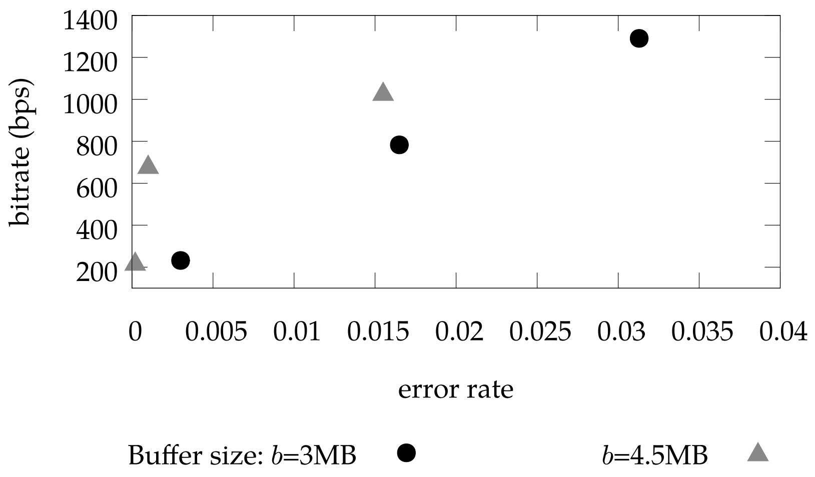

Figure 3.4: Bitrate as a function of the error rate, for two sizes of buffer *b*. *Laptop* setup (3MB LLC) in native environment.

decreases when the size of the last-level cache increases. This is expected since it takes longer to send a '1', as the sender must write into the entire last-level cache. Compared to the *laptop* setup, the error rate and the standard deviation have also increased. There are two factors that can explain these results. First, the ratio of the associativity over the number of cores is smaller in the *workstation* setup, which means that lines have a greater probability of being evicted by processes running in other cores, leading to a higher error rate. Second, the last-level cache is bigger in the *workstation* setup, which means that the allocation of a buffer might not cover all the sets of the last-level cache, leading to a difference in the error rate between runs, and thus a higher standard deviation.

3.6.3 Virtualized environment

We evaluate C5 in the *laptop* setup, using Xen 4.4 hypervisor. We run the sender as an unprivileged process in a guest virtual machine and the receiver as an unprivileged process in another guest virtual machine. The host and two guests run Ubuntu 14.04. We use the same operating system as in the native environment experiments to remove additional influence from the operating system. Each guest has one vCPU, and the host uses the default algorithm to schedule guest virtual machines.

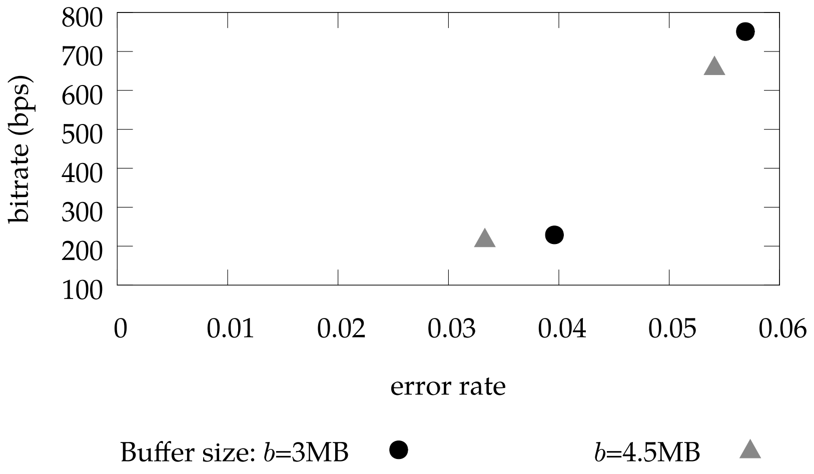

Figure 3.5: Bitrate as a function of the error rate, for two sizes of buffer b. *Laptop* setup (3MB LLC) in virtualized environment.

Figure 3.5 presents the results for two values of b (b = 3MB and $b = 4.5$MB), and two waiting time w (w = 4000µs and w = 1000µs). For b = 3MB (the size of the LLC), we obtain a bitrate between 229bps and 751bps by varying w. When we increase b to 4.5MB, the bitrate goes from 219bps to 661bps. There is a performance degradation compared to the native setup, but the bitrate stays in the same order of magnitude. The error rate is slightly higher than in the native setup, between 3.3% ($\sigma = 0.019$) and 5.7% ($\sigma = 8.8 \times 10^{-3}$), and is comparable for the two values of b. The standard deviation of the error rate is also higher than in the native setup, and is higher for a low value of b. This can be explained by the noise induced by virtualization, due to the hypervisor, the host and the two guest virtual machines.

3.7 Discussion

Reducing the number of targeted sets C5 evicts the whole last-level cache, thus evicting all sets. It is possible to reduce the number of targeted sets, even without knowing the complex addressing function. Indeed, the lower bits (12 bits for 4kB pages) of a virtual address are the same as the lower bits of its physical address. As a result, two processes can agree on a value for these bits. The sender thus targets one set every 64 sets, making less memory accesses to transmit a message. We implemented it and indeed found a speedup of around 64 times compared to the bitrate of C5, with a new bitrate of 23kbps. However, we did not properly evaluate the error rate.[2]

Signal quality Evicting cache lines by reading or writing memory modifies the quality of the signal. A read operation is generally less costly than a write operation, so the sender could prefer to perform reads instead of writes in this covert channel. However, reads do not have the same impact in terms of timing for the receiver. When the receiver loads a new cache line, there are two cases. In the first case, the line had previously only been read by the sender, and not modified. The receiver thus requires a single memory access to load the line. In the second case, the line had previously been modified by the sender. At the time of eviction, it needs to be written back in memory. This requires an additional memory access, and thus takes longer. We implemented both approaches and we choose to evict cache lines using memory writes because of the higher latency, which improves the signal quality.

Running on the same core Whenever the sender and the receiver run on the same core, C5 may benefit from optimizations. In this case, there is no need to evict the entire last-level cache: evicting the L1 cache that is 32kB is sufficient and faster than evicting 3MB. However, the sender and the receiver are scheduled by the OS or the hypervisor, and frequently run on different cores [XBJ$^+$11]. We would need a method to detect when both run on the same core, and adapt the sender algorithm accordingly. We preferred to keep our method simpler, less error-prone and agnostic to scheduling.

Detection algorithm The detection algorithm of the receiver impacts the overall bitrate. We put the priority on having a good detector at the receiver end, to minimize the error rate. In our implementation, the sender is waiting between the transmission of consecutive bits. The receiver uses a density-based clustering algorithm to separate the bits. Further work can be dedicated to reduce or eliminate the waiting time on the sender side, by using a different detection algorithm on the receiver side. The detection algorithm is nevertheless not the bottleneck of C5, so we do not expect a dramatic bitrate improvement by changing it.

[2]This improvement was suggested to us by Yossef Oren after we published the camera ready of [MNHF15]. It is also used in [OKSK15].

Different cache design and system architecture Our covert channel depends on the cache design. In particular, it critically relies on the shared and inclusive last-level cache. We believe this is a reasonable assumption, given that the latest generations of microarchitectures of Intel processors all have this design. The caches of AMD processors have historically been exclusive, and our covert channel is likely not to work with this cache design. However, inclusive caches seem to be a recent trend even at AMD. Indeed, the low-power microarchitecture named Jaguar introduced in 2015, and the future CPU microarchitecture named Zen both use an inclusive last-level cache. As Wu et al. [WXW12] noted, cache-based covert channels also need to be on the same processor. On a machine with two processors, two virtual machines are on average half the time on the same processor. Such a setting would introduce transmission errors. These errors may be handled by implementing error correcting codes or some synchronization between the receiver and the sender. In any case the bitrate would be reduced for C5 as well as for other cache-based covert channels.

Cloud computing setup In a public cloud setup such as on Amazon EC2, several factors may impact the performance and applicability of our covert channel. First, the sender and receiver must be co-resident, *i.e.*, they must run on the same hardware and hypervisor despite the virtual machine placement policy of the cloud provider. Section 2.3.1 provides some background on how to achieve and detect co-residency in cloud environments. In practice, it is costly and difficult to achieve. Second, the hypervisor, and in particular its associated vCPU scheduler, may have an impact on performance, as previously reported by [RTSS09, XBJ$^+$11]. Amazon EC2 relies on a customized version of Xen which uses an unknown scheduler. As a consequence, the results obtained in our lab experiments using Xen's default scheduler cannot be translated *as is* to a cloud setting. We expect the performance to be degraded in a cloud environment.

Noise induced by non-participating workload Similarly, we expect the error rate to increase in presence of a high non-participating workload, as it is the case with other cache covert channels [XBJ$^+$11]. The resulting error rate depends on the memory footprint of the workload, the core on which it executes, and on its granularity of execution compared to the transmission delay of the message.

3.8 Countermeasures

Section 2.4.7 described the state-of-the-art in terms of cache attacks countermeasures. In this section, we review some of these solutions, both at the hardware and at the software levels, to determine if they also apply to the mitigation of our covert channel.

3.8.1 Hardware countermeasures

At the hardware level, some countermeasures are designed to protect a particular sensitive application, e.g., a cryptographic algorithm. For example, PLcache [WL07] locks the cache lines of sensitive programs, to prevent their eviction by an attacker. Such countermeasures do not prevent covert channels, where both programs cooperate to exchange information and are controlled by an attacker.

Other countermeasures apply globally, they thus have more likelihood to thwart covert channels. Domnister et al. [DJL$^+$11] proposed to modify the replacement policy in the cache controller. Their cache design prevents a thread from evicting lines that belong to other threads. Although L2/L3 attacks and defense are out of the scope of their paper, if the policy is applied to the last-level cache, the sender cannot evict all lines of the receiver, so it may partially mitigate our covert channel too. However, performance degradation of this method on L1 cache is about 1% on average, up to 5% on some benchmarks. This mitigation might impact even more performance if also applied to the last-level cache. RPcache [WL07] randomizes the interferences such that information about cache timings is useless to an attacker. It is done by creating permutations tables so that the memory-to-cache-sets mapping is not the same for sensitive programs as for others. However, C5 is agnostic to this mapping since it targets the whole cache, so this solution may not mitigate our covert channel. These hardware-based solutions are currently not implemented.

Finally, exclusive caches, used by current AMD processors, mitigate our covert channel as they prevent the last-level cache from invalidating sets of private L1 caches. However, AMD has already shifted from exclusive to inclusive caches in their low-power microarchitectures, and has announced that their next CPU microarchitecture will also use an inclusive last-level cache. It is thus unlikely to see future generations of CPUs with an exclusive last-level cache.

3.8.2 Software countermeasures

Similarly to hardware countermeasures, software countermeasures that target applications are not suited to defend against covert channels, as the sender and the receiver are controlled by an attacker.

At the system level, the operating system or the hypervisor can take measures against last-level cache covert channels by isolating better the programs or virtual machines. Similar to Kim et al. [KPMR12], the hypervisor can partition the last-level cache such that each virtual machine works on dedicated sets within the last-level cache. This way, the sender cannot evict the lines of the receiver that is running in a different virtual machine.

Another countermeasure is to introduce noise. Düppel [ZR13] is a solution that repeatedly cleans the L1 cache. This introduces noise on the timing measurements of the attacker, thus rendering them useless. As the mitigation is only performed on the L1 cache, it may not mitigate our covert channel that exploits the last-level cache. Furthermore, applying this countermeasure to the whole cache hierarchy would lead to an important performance penalty, as this would nullify the purpose of the cache. Modifying the scheduler to introduce noise in the measurements as in [VRS14] can in practice complicate the transmission between the sender and the receiver, and thus reduce the bitrate.

3.9 Related work

Covert channels using caches have been known for a long time. Hu [Hu92] in 1992 was the first to consider the use of cache to perform cross-process leakage via covert channels. Covert channels in the cloud were introduced by Ristenpart et al. [RTSS09] in 2009, and were thus performed on older generations of processors. In particular, it was not possible to perform a cross-core channel using the cache. Ristenpart et al. built a covert channel for Amazon EC2, based on L2 cache contention that uses a variant of Prime+Probe [Per05, OST06]. Despite its low bitrate of 0.2bps, this covert channel shows deficiencies in the isolation of virtual machines in Amazon EC2. However, this covert channel has some limitations: the sender and receiver must synchronize and share the same core. Xu et al. [XBJ$^+$11] quantified the achievable bitrate of such a covert channel: they reached 215bps in lab condition, but only 3bps in the cloud. The dramatic drop is due to the fact that the covert channel does not work across cores, and thus the channel design has to take into account core migration.

Wu et al. [WXW12] designed a data transmission scheme purely time-based, for which the sender and receiver are not scheduled in a round-robin fashion, but simultaneously. The sender and the receiver can thus run on different cores. This covert channel also uses a variant of Prime+Probe. However, the sender and receiver have to agree on a set to work on, which ignores the addressing issue. Their experiment has been tested on a non-virtualized environment, and on a CPU with an older microarchitecture that does not feature complex addressing. They further assumed that cache-based covert channels are impractical due to the need of a shared cache. Yet, modern processors – including those used by Amazon EC2 – have all the properties that make cache-based covert channels practical, and thus this assumption needs to be revisited. Moreover, complex addressing on the last-level cache is now a common feature. In contrast with this work, we leverage the properties of modern hardware to build a covert channel that works across cores in a virtualized environment.

Flush+Reload attacks [YF14, IIES14b] rely on deduplication offered by the hypervisor. With deduplication, common pages use the same cache lines. These attacks thus bypass the uncertainty brought both by virtualization and complex addressing. They exploit the inclusive feature of last-level caches, and use the `clflush` instruction that flushes a line from the whole cache hierarchy. However, using deduplication imposes constraints on the platform where the attack can be performed. For instance, to the best of our knowledge, the Xen version used in Amazon EC2 does not allow deduplication. In contrast with these papers, we tackle the *addressing uncertainty* without any shared memory, which makes our covert channel possible in more restrictive environments.

Hund et al. [HWH13] reverse-engineered the complex addressing function in order to circumvent the kernel space ASLR. While this is a first step to resolve the addressing uncertainty brought by complex addressing on modern processors, the authors only reversed the function for a given Sandy Bridge processor. It is unknown if the function differs for processors of the same microarchitecture, or for processors of a different microarchitecture. Our covert channel is agnostic to this function, hence it applies to a large range of modern processors.

Concurrently to our work, Liu et al. [LYG$^+$15] demonstrated a Prime+Probe attack on the last-level cache that bypasses both virtualization and complex addressing, without any shared memory. Instead of reverse-engineering the complex addressing function, or targeting the whole last-level cache, they used a timing attack to find set of addresses that are located in the same last-level cache set. They used huge pages to resolve the uncertainty brought by memory virtualization. They demonstrated two cryptographic side channels, as well as a covert channel that has a bitrate of 1.2Mbps, with an error rate of 22%, on Sandy Bridge processors. Irazoqui et al. [IES15b] also used huge pages to resolve the uncertainty brought by memory virtualization, but they demonstrated a side channel on a Nehalem processor that does not use complex addressing.

3.10 Conclusions and perspectives

This chapter described C5, a novel covert channel that transfers messages across different cores of the same processor. Our covert channel tackles *addressing uncertainty* that is in particular introduced by hypervisors and complex addressing. In contrast to previous work, our covert channel does not require any shared memory. All these properties make our covert channel fast and practical.

We analyzed the root causes that enable this covert channel, *i.e.*, microarchitectural features such as the shared last-level cache, and the inclusive feature of the cache hierarchy. We experimentally evaluated our covert channel in native and virtualized environments. We successfully established a covert channel between virtual machines despite the CPU scheduler of the hypervisor. We measured a bitrate one order of magnitude above previous cache based covert channels in the same setup.

Concurrent work [LYG$^+$15] showed that it is possible to target precise sets on Sandy Bridge processors despite virtualization and complex addressing. However, we later showed [GMM15] that the new replacement policy in Ivy Bridge and Haswell requires an adaptation of the eviction strategy. Our covert channel has been demonstrated on an Ivy Bridge, and thus defeats this replacement policy, however it is not fine-grained enough to be used in a cryptographic side channel. The impact of this new replacement policy on side channels will be explored in a future work.

4

Reverse-engineering last-level cache complex addressing

Contents

4.1 Introduction

Cache attacks can operate at all cache levels: L1, L2 or last-level cache. Attacks on the L1 or L2 cache restrict the attacker to be on the same core as the victim. This is a too strong assumption on a multi-core processor when the attacker and the victim migrate across cores [RTSS09, XBJ^{+}11]. We thus focus on cache attacks on the last-level cache, which is shared among cores in modern processors. Attacks on last-level cache are more powerful as the attacker and the victim can run on different cores, but they are also more challenging. Without using any shared memory, an attacker has to find addresses that map to the same set, and exploit the cache replacement policy to evict lines. He faces two issues: the last-level cache is physically addressed, and modern processors map physical addresses to slices using the so-called *complex addressing* scheme which is undocumented. We detailed in the last chapter a method to bypass this complex addressing scheme, by evicting the whole last-level cache. This is a practical way to perform cache covert channels as we showed, but it is not fine-grained enough to perform, e.g., cryptographic side channels. Additionally, previous approaches already manually reverse-engineered the complex addressing function for specific Sand Bridge processors [HWH13, Sea15b]. However, they used a timing attack, that is unable to retrieve all the bits, leading to a partial function only. It is also unknown if the function is the same for other processors.

In this chapter, we present a novel and automatic method to reverse-engineer the addressing function of the last-level cache. We use hardware performance counters to monitor events associated with slices in order to map physical addresses to each slice. We thus recover all the bits of the function. We evaluated our method on a wide variety of processors, encompassing Sandy Bridge, Ivy Bridge and Haswell microarchitectures, for different numbers of cores.

Section 4.2 gives background on hardware performance counters. Section 4.3 shows how to use performance counters to derive a mapping between physical addresses and last-level cache slices. Section 4.4 details how we derive a compact addressing function for processors that have 2^n cores. Section 4.5 demonstrates how the function can be used in practice, by building a covert channel that is faster than C5, and exploiting the Rowhammer vulnerability in JavaScript. We discuss the differences between our function and the ones previously retrieved in Section 4.6, and related work in Section 4.7. Finally, Section 4.8 concludes and gives perspective on future work.

4.2 Hardware performance counters

Hardware performance counters are special-purpose registers that are used to monitor hardware-related events. Such events include cache misses or branch mispredictions, making the counters useful for performance analysis or fine tuning. Because performance counters require high level of privileges, they cannot be directly used for an attack.

The registers are organized by performance monitoring units (called PMON). Each PMON unit has a set of *counter registers*, paired with *control registers*. Performance counters can only be used to measure global events that happen at the hardware level, and not for a process in particular. This adds noise and has to be considered when performing a measurement.

There is one PMON unit, called CBo (or C-Box), per last-level cache slice. Each CBo has a separate set of counters, paired to control registers. Among available events, `LLC_LOOKUP` counts all accesses to the last-level cache. A mask on the event filters the type of the request (data read, write, external snoop, or any) [Int12, Int14c, Int14d].

Performance counters depend on the processor, but the CBo counters and the `LLC_LOOKUP` event are present in a wide range of processors, and documented by Intel.[1] Some adaptations are needed between different types of processors. Indeed, for Xeon Sandy Bridge, Xeon Ivy Bridge, Xeon Haswell and Core processors, the MSR addresses and the bit fields (thus the values assigned to each MSR) vary, but the method remains similar. Details of the MSR addresses and values can be found in Appendix C. Reading and writing MSR registers needs to be done by the kernel via the privileged instructions `rdmsr` and `wrmsr`.

4.3 Mapping physical addresses to cache slices

In this section, we present our technique for reverse-engineering the complex addressing function, using performance counters. Our objective is to build a table that maps a physical address (for each line of cache) to a slice (e.g., Table 4.1).

Algorithm 3 describes our technique. First, monitoring the `LLC_LOOKUP` event is set up by writing to *control registers* with `wrmsr`. Then, one memory address is repeatedly accessed (Listing 4.1) to generate activity on the corresponding slice. The *counter registers* are then read for each slice (each CBo). Next, the virtual address is translated to a physical address by reading the file `/proc/pid/pagemap`. Finally, the physical address is associated to the slice that has the most lookups. Such monitoring sessions are iterated with different addresses to obtain a set of pairs (physical address, slice), forming a table.

The number of times the address needs to be polled is determined experimentally to differentiate the lookup of this particular address in a slice from the noise of other last-level cache accesses. We empirically found that polling an

[1]For the Xeon range (servers): processors of the microarchitecture Sandy Bridge in [Int12], Ivy Bridge in [Int14c], and Haswell in [Int14d]. For the Core range (mobiles and workstations), in [Int14b] for the three aforementioned microarchitectures.

Table 4.1: Mapping table obtained after running Algorithm 3. Each address has been polled 10000 times. The columns *CBo 0* to *CBo 3* are obtained from experiment and the column *slice* is derived from it.

Physical address	**CBo 0**	**CBo 1**	**CBo 2**	**CBo 3**	**Slice**
0x3a0071010	**11620**	1468	1458	143	0
0x3a0071050	626	**10702**	696	678	1
0x3a0071090	498	567	**10559**	571	2
0x3a00710d0	517	565	573	**10590**	3
...	...	...	...	...	...

Algorithm 3 Constructing the address to slice mapping table.

```
1: $mapping \leftarrow$ new table
2: for each addr do
3:   for each slice do
4:     write MSRs to set up monitoring LLC_LOOKUP event
5:   end for
6:   polling(addr)                          // see Listing 4.1
7:   for each slice do
8:     read MSRs to access LLC_LOOKUP event counter
9:   end for
10:  $paddr \leftarrow$ translate_address(addr)
11:  find slice $i$ that has the most lookups
12:  insert ($paddr$, $i$) in $mapping$
13: end for
```

Listing 4.1 Memory polling function.

```
void polling(uintptr_t addr){
  register int i asm ("eax");
  register uintptr_t ptr asm ("ebx") = addr;
  for(i=0; i<NB_LOOP; i++){
    clflush((void*)ptr);
  }
}
```

address $10\,000$ times is enough to distinguish the correct slice from noise without ambiguity, and to reproduce the experiment on different configurations. The polling itself is carefully designed to avoid access to memory locations other than the tested address (see Listing 4.1). To this end, most of the variables are put in registers, and the only access to main memory is performed by the `clflush` instruction that flushes the line in all cache hierarchies. The `clflush` instruction causes a lookup in the last-level cache even when the line is not present.

Table 4.2 shows the characteristics of the CPUs we tested. Scanning an address per cache line, *i.e.*, an address every 64B, takes time, but it is linear with the memory size. Scanning 1GB of memory takes a bit less than 45 minutes. We now estimate the storage cost of the mapping table. The lowest 6 bits of the address are used to compute the offset in a line, hence we do not need to store them. In practice, it is also not possible to address all the higher bits because we are limited by the memory available in the machine. For a processor with c slices, the slice is represented with $\lceil log_2(c) \rceil$ bits. A configuration of, e.g., 256GB ($= 2^{38}$) of memory and 8 cores can be represented as a table with an index of 32 ($= 38 - 6$) bits. Each table entry contains 3 bits identifying the slice and an additional bit indicating whether the address has been probed or not. The size of the table is thus $2^{32} \times 4$ bits = 2GB.

Note that the attacker does not necessarily need the entire table to perform an attack. Only the subset of addresses used in an attack is relevant. This subset can be predefined by the attacker, e.g., by fixing the bits determining the set. Alternatively, the subset can be determined dynamically during the attack, and the attacker can query an external server to get the corresponding slice numbers.

4.4 Building a compact addressing function

4.4.1 Problem statement

We aim at finding a function, as a compact form of the table. The function takes n bits of a physical address as input parameters. In the remainder, we note b_i the bit i of the address. The function has an output of $\lceil log_2(c) \rceil$ bits for c slices. To simplify the expression and the reasoning, we express the function as several Boolean functions, one per bit of output. We note $o_i(b_{63}, \ldots, b_0)$ the function that determines the bit i of the output.

Table 4.2: Characteristics of the Intel CPUs used in our experimentations (mobile and server ranges).

Name	**Model**	**μ-arch**	**Cores**	**Mem**
config_1	Xeon E5-2609 v2	Ivy Bridge	4	16GB
config_2	Xeon E5-2660	Sandy Bridge	8	64GB
config_3	Xeon E5-2650	Sandy Bridge	8	256GB
config_4	Xeon E5-2630 v3	Haswell	8	128GB
config_5	Core i3-2350M	Sandy Bridge	2	4GB
config_6	Core i5-2520M	Sandy Bridge	2	4GB
config_7	Core i5-3340M	Ivy Bridge	2	8GB
config_8	Core i7-4810MQ	Haswell	4	8GB
config_9	Xeon E5-2640	Sandy Bridge	6	64GB

Our problem is an instance of Boolean function minimization: our mapping can be seen as a truth table, that can consequently be converted to a formula in Disjunctive Normal Form (DNF). However, the minimization problem is known as NP-hard, and is thus computationally difficult [CH11].

Existing work on Boolean function minimization does not seem suitable to reconstruct the function from this table. Exact minimization algorithms like Karnaugh mapping or Quine-McCluskey have an exponential complexity in number of input bits. In practice those are limited to 8 bits of input, which is not enough to compute a complete function. The standard tool for dealing with a larger number of inputs is Espresso, which relies on non-optimal heuristics. However, it does not seem suited to handle truth tables of hundreds of millions of lines in a reasonable time.[2] It also gives results in DNF, which will not express the function compactly if it contains logical gates other than AND or OR. Indeed, we provided lines for a subset of the address space to Espresso, but the functions obtained were complex and we did not succeed to generalize them manually. They were generated from a subset, thus they are only true for that subset and do not apply to the whole address space.

We thus need hints on the expression of the function to build a compact addressing function. We did this by a first manual reconstruction, then followed by a generalization. We have done this work for processors with 2^n cores, which we consider in the remainder of the section.

[2]We let Espresso running for more than 2000 hours without any results on a table of more than 100.000.000 lines, which only represents the sixth of the 64GB of memory of the machine.

4.4.2 Manually reconstructing the function for Xeon E5-2609 v2

We now explain how one can manually reverse-engineer a complex addressing function: this is indeed how we started for a Xeon E5-2609 v2 (*config_1* in Table 4.2). In Section 4.4.3, we will explain how this can be automated and generalized to any processor model with 2^n cores. The following generalization removes the need to perform manual reconstruction for each setup.

We manually examined the table to search patterns and see if we can deduce relations between the bits and the slices. We performed regular accesses to addresses which were calculated to fix every bit but the ones we want to observe, e.g., regular accesses every 2^6 bytes to observe address bits $b_{11} \dots b_6$. For bits $b_{11} \dots b_6$, we can observe addresses in 4kB pages. For the higher bits (b_{12} and above) we need contiguous physical addresses in a bigger range to fix more bits. This can be done using a custom driver [HWH13], but for implementation convenience we used 1GB pages. Across the table, we observed patterns in the slice number, such as the sequences (0,1,2,3), (1,0,3,2), (2,3,0,1), and (3,2,1,0). These patterns are associated with the XOR operation of the input bits, which made the manual reconstruction of the function easier.

We obtained these two binary functions:

$$\begin{aligned} o_0(b_{63}, \dots, b_0) = b_6 \oplus b_{10} \oplus b_{12} \oplus b_{14} \oplus b_{16} \oplus b_{17} \oplus b_{18} \oplus b_{20} \oplus b_{22} \\ \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{32} \oplus b_{33}. \end{aligned}$$

$$\begin{aligned} o_1(b_{63}, \dots, b_0) = b_7 \oplus b_{11} \oplus b_{13} \oplus b_{15} \oplus b_{17} \oplus b_{19} \oplus b_{20} \oplus b_{21} \oplus b_{22} \\ \oplus b_{23} \oplus b_{24} \oplus b_{26} \oplus b_{28} \oplus b_{29} \oplus b_{31} \oplus b_{33} \oplus b_{34}. \end{aligned}$$

We confirmed the correctness of the obtained functions by comparing the output of the slice calculated with the function against the entire mapping table obtained with the MSRs.

4.4.3 Reconstructing the function automatically

Our manual reconstruction shows that each output bit $o_i(b_{63}, \dots, b_0)$ can be expressed as a series of XORs of the bits of the physical address. Hund et al. [HWH13] manually reconstructed a mapping function of the same form, albeit a different one. In the remainder, we thus hypothesize, and subsequently validate the hypothesis, that the function has the same form for all processors that have 2^n cores.

Table 4.3: Functions obtained for the Xeon and Core processors with 2, 4 and 8 cores. Gray cells indicate that a machine with more memory would be needed to determine the remaining bits.

		Address bit																													
		37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06
2 cores	o_0	▨	▨	▨	▨	▨	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$
4 cores	o_0	▨	▨	▨		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$
	o_1	▨	▨	▨	$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$	
8 cores	o_0		$\oplus$	$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$
	o_1	$\oplus$		$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$		$\oplus$		$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$		$\oplus$		$\oplus$		$\oplus$		$\oplus$				$\oplus$	
	o_2	$\oplus$	$\oplus$	$\oplus$	$\oplus$			$\oplus$	$\oplus$			$\oplus$	$\oplus$			$\oplus$	$\oplus$			$\oplus$			$\oplus$			$\oplus$	$\oplus$				$\oplus$		

The fact that the function only relies on XORs makes its reconstruction a very constrained problem. For each Boolean function $o_i(b_{63}, \ldots, b_0)$, we can analyze the implication of the address bits independently from each other, in order to access only a handful of physical addresses. Our algorithm finds two addresses that only differ by one bit, finds their respective slices using performance counters, and compares the output. If the output is the same, it means that the bit is not part of the function. Conversely, if the output differs, it means that the bit is part of the function. Note that this only works for a XOR function. This algorithm is linear in number of bits.

To implement the algorithm, we use huge pages of 1GB on Xeon processors (resp. 2MB on Core processors), which is contiguous physical memory naturally aligned on the huge page size. The offset in a huge page is 30-bit (resp. 21-bit) long, therefore the lowest 30 bits (resp. 21 bits) in virtual memory will be the same as in physical memory. We thus calculate offsets in the page that will result in physical addresses differing by a bit, without converting virtual addresses to physical addresses. To discover the remaining bits, we allocate several huge pages, and convert their base virtual address to physical address to find those that differ by one bit. In order to do this, we allocate as many huge pages as possible.

To evaluate the algorithm, we retrieved the function for all models from *config_1* to *config_8* of Table 4.2. Results are summarized in Table 4.3. The functions are given for the machine that has the most memory, to cover as many bits as possible. We remark that the functions, for a given number of cores, are identical among all processors, for all ranges of products and microarchitectures. Using the aforementioned algorithm, we obtained those functions quickly (from a few seconds to five minutes in the worst cases). We also remark that we have in total 3 functions o_0, o_1 and o_2 for all processors, and that the functions used only depend on the number of cores, regardless of the microarchitecture or the product range. While in retrospective this seems to be the most straightforward solution to be adopted by Intel, this was far from evident at the beginning of our investigations. Now that the functions are known, an attacker can use them to perform his attack without any reverse engineering.

4.5 Applications

Reverse-engineering the complex addressing function is orthogonal to performing cache attacks. Indeed, knowing the correct addressing function can help any fine-grained attack on the last-level cache. Cache attacks rely on the attacker evicting data from a cache level. This can be done by the `clflush` instruction. However, it requires shared memory in a covert or side channel scenario, and it is not available in all environments. We thus focus on building attacks without this instruction. To perform an attack on the last-level cache, the attacker needs to create an eviction set, and to subsequently access the data to evict the lines that are currently cached.

There are two methods to create an eviction set: a *dynamic* approach based on a timing attack that does not require the function, and a *static* approach that uses the function to compute addresses that belong to an eviction set. Building a static eviction set has the advantage of being faster than building a dynamic one. Indeed, the function is already known, whereas the dynamic set has to be computed for each execution. Moreover, in [GMM15] we show that dynamically computing a set to achieve an optimal eviction is a slow operation.

Several publications already used a static approach to perform cache attacks. Hund et al. [HWH13] defeated KASLR. Similarly, Irazoqui et al. [IES15b] performed a side channel on a Nehalem CPU that does not use complex addressing. Yet, their attack requires understanding the slice selection, and thus the complex addressing function for more recent CPUs. We now detail two new applications: the construction of a covert channel, and the implementation of the Rowhammer vulnerability in JavaScript.

4.5.1 Building a faster covert channel

To verify empirically the correctness of the function, we build a covert channel. This covert channel uses principles similar to C5 (see Chapter 3). It is based on the fact that the last-level cache is inclusive, *i.e.*, when a line is evicted from the last-level cache, it is also evicted from the L1 and L2. With this property, a

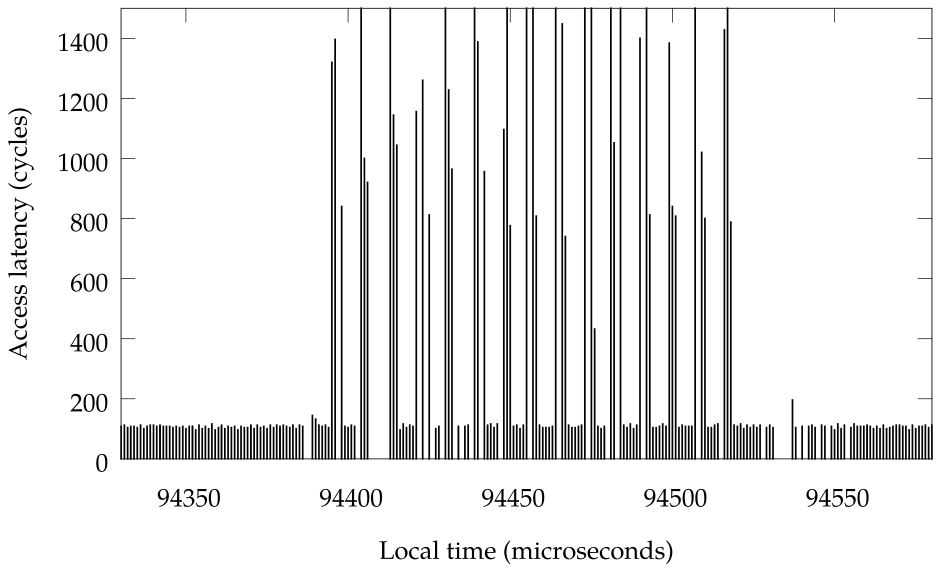

Figure 4.1: Receiving interleaved ‘0’s and ‘1’s to test the raw bitrate of the covert channel.

program on any core can evict a line from the private cache of another core. This property can then be used by two programs to communicate. C5 bypasses the complex addressing issue by evicting the whole last-level cache. However, the last-level cache typically stores a few megabytes, and thus the sender needs to access a buffer that is the size of (or bigger than) the last-level cache to evict it entirely. Having the complex addressing function, the sender targets a set in a slice, and thus evicts a cache line with much fewer accesses. For example, in the case of a 12-way associative last-level cache, assuming a pseudo-LRU replacement policy, the sender needs approximately only 12 accesses to evict the whole set.

In this covert channel, the sender creates a set of physical addresses that map to the same set, with a static approach using the function and the translation from virtual to physical addresses. It repeatedly accesses these addresses to send a ‘1’, and does nothing to send a ‘0’. The receiver has a set of physical addresses that map to the same last-level cache set as the sender’s. When the sender sends a ‘1’, it evicts the data of the receiver from the last-level cache, and thus from its private L1 cache. The receiver consequently observes a slow access to its set.

We conduct an experiment on *config_1* to estimate the bitrate of this covert channel, in which the sender transmits interleaved '0's and '1's. Figure 4.1 illustrates the measurements performed by the receiver. According to the measurements, 29 bits can be transmitted over a period of 130 microseconds, leading to a bitrate of approximately 223 kilobits per second. This is a speedup of around 300 times compared to C5. A more thorough evaluation is required to evaluate the error rate of this covert channel.

4.5.2 Exploiting the Rowhammer vulnerability in JavaScript

We also apply our findings to exploit Rowhammer [KNQ14] in restrictive environments.[3] Rowhammer is not a typical cache attack, since it exploits faulty DRAM to flip bits. This vulnerability has already been exploited [Sea15a] to gain root privileges and to evade a sandbox, showing the severity of faulting single bits for security. However, the vulnerability is triggered by frequent accesses to the DRAM, *i.e.*, non-cached accesses. The original exploits used the `clflush` instruction, that is not available in, e.g., JavaScript. An attack that seeks to avoid using the `clflush` instruction thus also needs to compute a set of addresses that evict the address that needs to be flushed.

Yet, the bit flips only occur when numerous accesses are performed between two consecutive refreshes of the row buffer. The challenge is thus to perform accesses that are solely non-cached quickly enough to induce bit flips. The computed eviction set must therefore be minimal in order to reduce the number of memory accesses.

First, to build a proof-of-concept in native environment without the `clflush` instruction, we computed eviction set using a static approach, with the complex addressing function. We show that it is possible to trigger bit flips this way, albeit significantly less than with the `clflush` instruction (see Figure 4.2). This makes the attack technique independent of the specific CPU microarchitecture, programming language and runtime environment, as long as the stream of memory accesses is executed quickly enough. The JavaScript proof-of-concept also uses a static approach, accompanied by a tool that spies Firefox memory mappings. The final attack in JavaScript uses a dynamic approach to compute eviction sets, as the attack cannot rely on the virtual to physical address translation.

[3]This work is the result of a collaboration with Daniel Gruss and Stefan Mangard. It is described in more detail in an article [GBM15] of which I am a co-author.

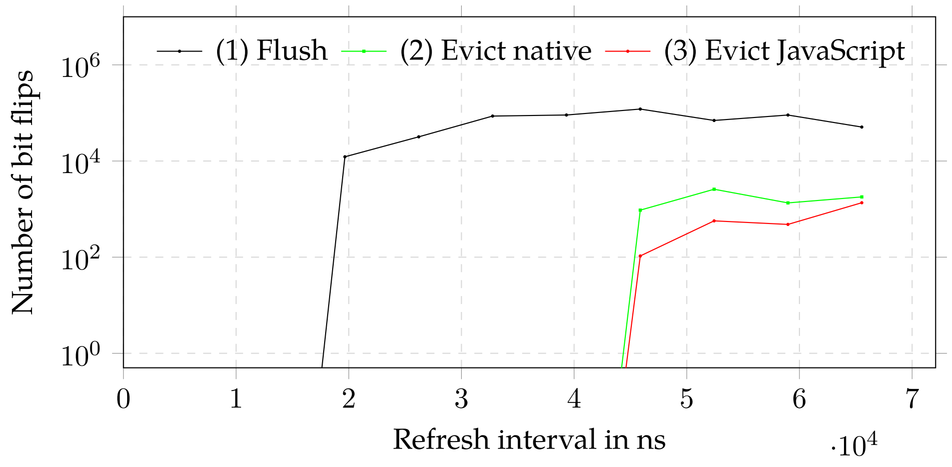

Figure 4.2: Number of bit flips as a function of the DRAM refresh interval, within 15 minutes, in different configurations: (1) in native environment with `clflush`, (2) in native environment with an eviction set (without `clflush`), (3) in JavaScript (from [GMM15]).

4.6 Discussion

4.6.1 Dealing with unknown physical addresses

The translation from virtual to physical addresses is unknown to the attacker in most practical setups, like in virtualized or sandboxed environments. We now describe a possible extension to the covert channel described in Section 4.5.1 to avoid using this address translation.

Similarly to the work of Liu et al. [LYG+15] and Irazoqui et al. [IES15b], the sender and the receiver both use huge pages. Cache index bits are thus the same for virtual and physical addresses. Using the function only on the bits in the offset of the huge page, the sender is able to create a set of addresses that map to the same set, in the same slice. As some bits of the physical address are unknown, he does not know the precise slice. However, he does know that these addresses are part of a single set, in a single slice.

The receiver now performs the same operation. The receiver only knows the index set to target, but he does not know in which of the n slices. He thus creates n sets of addresses, each one being in a different slice. He then continuously accesses each of these sets. The receiver will only receive transmitted bits in a single set: from now on, he can target a single set. The sender and the receiver are effectively accessing the same last-level cache set in the same slice.

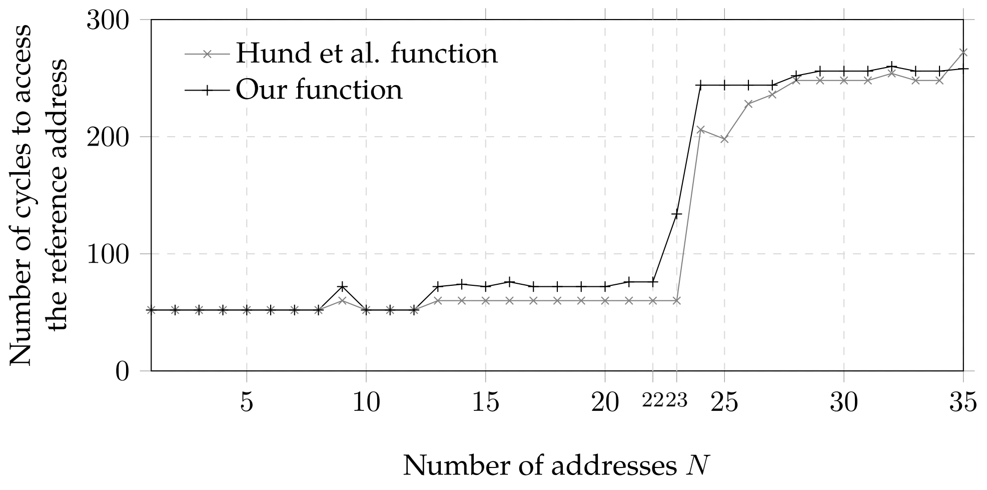

Figure 4.3: Median number of CPU cycles to access a reference address, after accessing N addresses in the same set, which is calculated using [HWH13] and our function. Results on 100 runs, on $config\_1$ (Ivy Bridge with a 20-way associative LLC).

4.6.2 Comparison to previously retrieved functions

We observe that the functions we obtained differ from the ones obtained by Hund et al. [HWH13], and Seaborn [Sea15b]. In particular, Hund et al. found that the functions only use the tag bits (bits b_{17} to b_{31}). We argue that their method does not infer the presence of the bits used to compute the set (bits b_6 to b_{16}). Indeed, as they searched for colliding addresses, they obtained addresses that belong to the same slice and the same set. As in this case the set is directly mapped to the bits b_6 to b_{16}, addresses that collide have the same values for these bits. Therefore, if the function that computes the slice uses the bits b_6 to b_{16}, the method of [HWH13] is not able to retrieve them. On the contrary, our method retrieves the slices regardless of the sets, leading to a complete function.

We also observe that the function we retrieved for 2-core CPUs is the same as the one retrieved in [Sea15b], albeit a more complete one. However, the function we retrieve for 4-core CPUs does not use the same bits as the one retrieved in [HWH13]. We argue that we do have access to the ground truth (*i.e.*, the slices accessed), whereas they rely on *indirect measurements*. Several correct functions can however be retrieved, as the slices can be labeled differently from one work to another.

To compare our function against [HWH13], we performed the following experiment. Using the retrieved addressing function, we constructed a set of physical addresses that are supposed to map the same set (thus the same slice). We accessed N different addresses from this set. We then measured the access time to the first reference address accessed, to see if it was evicted from the cache. Figure 4.3 shows the median number of CPU cycles to access the reference address for different values of N, for 100 runs. The function that is the most precise should have a memory access time spike the closest to $N = 20$ (which is the cache associativity). We observe that both functions have a spike slightly after $N = 20$. We note that the spike occurs for a value $N > 20$ and not exactly $N = 20$: it is most likely due to the fact that the replacement policy on Ivy Bridge is not strictly LRU, but a variant called Quad-Age LRU [JGSW]. In practice, both functions are able to evict a cache line with few accesses. However, our function seems more precise than the one of [HWH13], leading to fewer accesses to evict a cache line ($N = 23$ accesses for our function, $N = 24$ for [HWH13]), and a sharper transition. This also confirms the correctness of our function.

4.7 Related work

Hardware performance counters are traditionally used for performance monitoring. They have also been used in a few security scenarios. In defensive cases, they are used to detect an anomalous behavior such as malware detection [DMS+13], integrity checking of programs [MZK11], control flow integrity [XLCZ12], and binary analysis [WHF+12]. Uhsadel et al. [UGV08] used performance counters in offensive cases to profile the cache and derive a side-channel attack against AES. However, performance counters can only be read with high privileges, *i.e.*, in kernel-mode, or being root in user-mode if a driver is already loaded. Contrary to this attack, we use performance counters to infer hardware properties offline, and our subsequent cache attack does not need high privileges.

Liu et al. [LYG+15], and Oren et al. [OKSK15] have extended Prime+Probe attacks to the last-level cache. Even if we share the same motivation, *i.e.*, performing cache attacks on recent processors without any shared memory, our approach is different. Moreover, our work has a broader application, as it contributes to a better understanding of the undocumented complex addressing function, possibly leading to other types of attacks.

Other work is directly interested in retrieving the function, and several attempts have been made to reverse-engineer it. Hund et al. [HWH13] performed a manual reverse engineering for a 4-core Sandy Bridge CPU. Seaborn [Sea15b] continued the work of [HWH13], with a 2-core Sandy Bridge CPU. Both work use colliding addresses, *i.e.*, *indirect measurements*, to derive the function. Our method, using performance counters, performs *direct measurements*, *i.e.*, retrieves the exact slice for each access. We thus show that the functions in [HWH13, Sea15b] are partially incorrect, even though they are sufficient to be used in practice. We also derive a function for all processors with 2^n cores, automating reverse engineering. Different from these two approaches, we also have tested our method on a large variety of processors. Concurrently to our work, Irazoqui et al. [IES15c] worked on automating this reverse engineering, and evaluated their work on several processors. However, their method is similar to Hund et al. [HWH13], and thus suffers from the same limitations.

4.8 Conclusions and perspectives

In this chapter, we introduced a novel method to reverse-engineer Intel's undocumented complex addressing, using hardware performance counters. We showed that this method is able to retrieve more bits in the function than previous work, that used a timing attack to recover the function. We evaluated our approach on a wide range of processors, encompassing Sandy Bridge, Ivy Bridge and Haswell microarchitectures, and different numbers of cores. We showed that in the case of CPUs with 2^n cores, the functions are identical across microarchitectures, and only depend on the number of cores.

The reversed functions can be exploited by an attacker to target specific sets in the last-level cache. We built a covert channel that does not rely on shared memory, and target the cache sets, thus being more fine-grained and greatly improving C5 (see Chapter 3). We also directly applied our findings to exploit the Rowhammer vulnerability in a JavaScript environment, where the `clflush` instruction is not available. We conclude that current countermeasures that disable shared memory or the `clflush` instruction in sensitive environments are not effective.

Our work expands the understanding of these complex and only partially documented pieces of hardware that are modern processors. We foresee several directions for future work. First, a compact representation for CPUs with a number of cores different from 2^n would generalize our findings. Second, we believe that new attacks could be made possible by knowing the complex addressing of a cache. Finally, we believe that the understanding of the complex addressing function enables the development of countermeasures to cache attacks.

5

Information leakage on GPU memory

Contents

5.1 Introduction

GPUs benefit from a great interest from the scientific community. Indeed, GPGPU allows performing massively parallel general purpose computations by leveraging the inherent parallelism of GPUs. The applications are diverse, such as finance [BSS10, KP05, TB10], encryption [gE, Yam07], and Bitcoin mining [Kol]. GPUs have recently been offered by several cloud computing providers to supply on demand and pay-per-use of otherwise very expensive hardware. Additionally, graphic-intensive applications using virtualization for rendering are emerging. Low-end devices such as tablets can be used to play video games on *cloud gaming* platforms. Regular desktops or laptops can be used to perform tasks such as movie editing or computer-aided design on virtualized workstations.

GPUs have been designed to provide maximum performance and throughput. However, they have not been designed for concurrent accesses, *i.e.*, to support virtualization or simultaneous users. It is known that GPU buffers are not zeroed when allocated [Ker12]. This raises confidentiality issues between different programs or different users when GPUs are used natively on personal computers [DLV13]. Clearly, the attack surface is larger in a cloud environment when several users exploit the same GPU one after another or even simultaneously. Yet, such a setup has not been previously studied. Moreover, identifying possible information leakage in such environments is an intricate problem due to two layers of obscurity: the cloud provider as well as the GPU.

In this chapter, we study information leakage on GPUs and evaluate its possible impact on GPU clouds. We systematically experiment and analyze the behavior of GPU global memory in non-virtualized and virtualized environments. In addition to previous work [BKS13, DLV13], we show how an attacker can retrieve information from GPU global memory using a variety of drivers and frameworks. Furthermore, we find that in the rare cases where GPU global memory is zeroed, it is only as a side effect of Error Correction Codes (ECC) and not for security reasons. We also propose a method to retrieve memory in a driver agnostic way that bypasses some memory cleanup measures a conscious programmer may have implemented.

Section 5.2 details our attacker model and Section 5.3 the security impact of the different GPU virtualization techniques. We detail our setup in Section 5.4. We show two techniques to access the memory: leveraging the GPGPU runtime (Section 5.5), and exploiting the PCI configuration space (Section 5.6). We discuss countermeasures in Section 5.7 and related work in Section 5.8. We conclude and give perspective on future work in Section 5.9.

5.2 Attacker model

The objective of the attacker is to learn some information about the victim. This can occur directly by retrieving data owned by the victim in the GPU memory, or indirectly through side channels. We assume that the attacker has full control over a virtual machine. In our case, the virtual machine has access to a virtualized GPU. We consider two cases. First, the *serial attacker* has access to the same GPU as the one of the victim, before or after the victim. He will seek for traces of data previously left in different GPU memory levels. The experiments in the remainder of this chapter consider this particular attacker. Second, the *parallel attacker* and the victim are running simultaneously on the same virtualized GPU. He may also have direct access to memory used by the victim, if memory management is not properly implemented. However, as the *parallel attacker* shares the device with the victim, he may also abuse some side channels on the GPU, possibly allowing him to recover useful information.

The serial attacker can have access to the GPU memory in two different ways. In our experiments, we outline two types of attacks that require different capabilities for the attacker and differ in their results:

– In the first scenario, the attacker accesses portions of the GPU memory through a GPGPU runtime. He does not need root privileges since he uses perfectly legitimate calls to the CUDA runtime API.

– In the second scenario, the attacker accesses the GPU memory through the PCI configuration space. We assume the attacker has root privileges, either because he controls the machine or because he compromised it by exploiting a known privilege escalation. This attack assumes a more powerful attacker, but gives a complete snapshot of the GPU memory.

5.3 Impact of the GPU virtualization techniques on security

We presented different solutions to virtualize I/O devices in Section 2.2.3, which apply to GPUs. We now review their impact on security.

5.3.1 Emulation

Emulation is conceptually the safest virtualization technique. It is the one that brings the most interposition, *i.e.*, the hypervisor is able to inspect, and possibly modify or deny, all guests calls. Emulation also implements a narrow API, which limits the attack surface. Emulation often does not rely on actual hardware. Therefore, information leakage – or side channels – that is due to hardware sharing is effectively eliminated.

5.3.2 Split driver model

The split driver model is prone to information leakage and side channels enabled by shared hardware. Furthermore, the backend driver has to ensure the isolation of guests that share the same hardware. GPU drivers have not been designed with that goal in mind, therefore, the backend driver should completely be redesigned to address this. From an isolation, interposition and attack surface perspective, the split driver model is between emulation and direct device assignment. The API exposed to the guest domain is limited, which makes the split driver model a safe approach at first sight.

Nevertheless, if the backend driver runs on the privileged domain and not in a separate isolated driver domain, the device driver is part of the Trusted Computing Base (TCB), along with the hypervisor and the hardware. As such, a compromise of the backend driver can lead to the compromise of the entire system and break isolation between guest virtual machines. Reducing the TCB to its minimum is a common method to improve security. One approach is [Smo09], that breaks the monolithic Gallium 3D graphic driver to move a portion of the code out of the privileged domain. More generally, reducing the TCB is a daunting task given that the TCB of a virtualization platform is already very large [CNZ^{+}11]. Drivers are well-known to be a major source of operating systems bugs [CYC^{+}01]. GPU drivers are also very complex, require several modules and have a large code base. In the case of NVIDIA drivers, code cannot be inspected and verified since it is closed source. Like any complex piece of software, GPU drivers can suffer from vulnerabilities, such as those reported for NVIDIA drivers [CVE12a, CVE12b, CVE13a, CVE13b, CVE13c].

5.3.3 Direct device assignment

Direct device assignment gives direct access to a physical GPU, with a very limited level of interposition. The PCI passthrough is managed by QEMU and the IOMMU, that become two targets for attacks. The attack surface of the IOMMU is large since it has to handle every calls to the hardware: Memory-Mapped Input/Output (MMIO), Programmed Input/Output (PIO), DMA, interrupts. Although a piece of hardware is generally known as more secure than a piece of software, the IOMMU is prone to attacks [LLND10, WR11]. Side channels are of less importance because the GPU is not simultaneously shared by two tenants, but information leakage can still occur given that it is physical hardware that is shared across different sessions.

5.3.4 Direct device assignment with SR-IOV

Subsequently to our study, NVIDIA launched GPUs that handle virtualization in hardware, called GRID. Amazon EC2 documents using GRID K520 boards in two types of GPU instances [Amab]. The `g2.2xlarge` instance offers 1 GPU, and the `g2.8xlarge` instance 4 GPUs [Amaa]. According to NVIDIA, the

GRID K520 boards are composed of 2 GK104 GPUs [NVI], and support up to 16 concurrent users. The `g2.2xlarge` instance thus most probably shares the board with another instance. To the best of our knowledge, no study has been conducted to assess the security of this setup. Because they are designed for virtualization and for sharing, it is likely that they provide an isolation mechanism that will prevent *direct* information leakage from a parallel attacker. However, if memory cleaning is not properly implemented, it is the same situation as direct device assignment for a serial attacker. Moreover, performance and resource sharing are antagonistic to side channel resistance. Therefore, we expect that *indirect* information leaks are possible.

Full emulation and split driver techniques have low maturity and performance, and SR-IOV GPUs were not deployed at the time of our study. Therefore, in the rest of this chapter, we focus on data leaks in virtualization setups when GPUs are used in direct device assignment mode and in cloud setups. This effectively restricts the attacker model to the *serial attacker*.

5.4 Experimental setup

In this section, we detail the experiments that we conducted during our study. We consider the *serial attacker*. We organize our experiments according to two main parameters: the degree of virtualization, and the method used to access the GPU memory.

We pursue experiments using no virtualization, and using virtualization with direct device assignment. We use a lab setup for both settings and a real life cloud computing setup using Amazon. In our virtualized lab setup, we test two hypervisors: KVM and Xen. For both of them, we used HVM virtualization, with VT-d enabled. The administrative and guest virtual machines run GNU/Linux. The cloud computing setup is an Amazon GPU instance that uses Xen HVM virtualization with an NVIDIA Tesla GPU in direct device assignment mode. The virtual machine also runs GNU/Linux.

We pursue experiments accessing the memory with different GPGPU frameworks under different drivers (see Section 2.5.2). We also access the memory with no framework through the PCI configuration space, in a driver agnostic way. To that extent, we build a generic CUDA *taint* program and two *search* programs, depending on the access method.

Table 5.1: Overview of the attacks and results. The different actions between *taint* and *search* are: (1) switch user; (2) soft reboot bare machine or VM; (3) reset GPU using `nvidia-smi` utility; (4) kill VM and start another one; (5) hard reboot machine. ✓ indicates a leak, and ✗ no successful leak. N/A means that the attack is not applicable.

		Actions between *taint* and *search*				
Setup	ECC	1	2	3	4	5
GPGPU runtime access						
Native	on	✓	✗	✗	N/A	✗
	off	✓	✓	✓		✗
Virtualized	on	✓	✗	✗	✗	✗
	off	✓	✓	✓	✓	✗
Cloud	on	✓	✗	✗	N/A[a]	N/A
	off	✓	✓	✓		
PCI configuration space access						
Native	on	N/A[b]	✗	✗	N/A	✗
	off		✓	✓		✗
Virtualized	–	N/A[b]	✗	✗	✗	✗
Cloud	–	N/A[b]	✗	✗	N/A[a]	N/A

[a]We cannot guarantee that we end up in the same physical machine after releasing a virtual machine in the cloud setup.

[b]The access through PCI configuration space needs root privilege.

1. *Taint* writes identifiable strings in the global memory of the GPU. It uses the CUDA primitives `cudaMalloc` to allocate space on the global memory, `cudaMemcpy` to copy data from host to device, and `cudaFree` that frees memory on the device.

2. *Search* scans the global memory, searching for the strings written by *taint*. The program that uses a GPGPU framework operates in the same way as *taint* by allocating memory on the device. However, data is copied from device to host before finally freeing memory. The other program uses PCI configuration space.

We first execute *taint*, then *search*, with various actions between these two executions. An information leakage occurred if *search* can retrieve data written by *taint*. Table 5.1 summarizes our experiments and their results.

5.5 Accessing memory through GPGPU runtime

In this section, we detail our method and results to access the GPU memory with CUDA and Gdev runtimes, in three environments: native, virtualized and cloud.

5.5.1 Native environment

We conduct experiments similar to [BKS13, DLV13] with a Quadro Fermi GPU that does not support ECC. We validate information leakage on two frameworks: (i) using the runtime API on top of the CUDA runtime and the NVIDIA driver and (ii) using the driver API on top of the Gdev runtime and the Nouveau driver. We observed information leakage when users switch, when there is a soft reboot and when the GPU is reset, *i.e.*, in all cases between *search* and *taint* except for the hard reboot. This indicates that the GPU maintains data in memory as long as it is powered, *i.e.*, anyone can retrieve data during this time. The driver and framework do not impact memory leakage in this setting.

We now consider a Tesla Kepler GPU which provides ECC for its memory. We found that the Tesla GPU has two options that impact the behavior of memory: the persistence mode, and ECC mode. Enabling *persistence* keeps the driver loaded even when no application is accessing the GPU and minimizes the driver load latency. When the *Error Correction Code* option is enabled, part of the dedicated memory is used for ECC bits – this reduces the available memory by 12.5%. ECC protects register files, L1/L2 caches, shared memory, and DRAM [NVI10]. It takes effect after the next reboot, or device reset.

Table 5.2: Information leakage with user switch between the execution of *taint* and *search*, as function of ECC and persistence mode. Tested on a Tesla card in a native environment. ✓ indicates a leak, and ✗ no successful leak.

	ECC enabled	**ECC disabled**
Persistence off	✗	✓
Persistence on	✓	✓

Table 5.2 shows in which cases we could observe information leakage with a user switch on the Tesla Kepler GPU in a native environment. The only case where we could not observe any information leakage is when ECC is enabled and persistence is disabled. In this mode, the driver loads dynamically each time a GPU application is executed. These experiments suggest that memory cleaning is triggered by loading the driver when ECC is enabled. Furthermore, memory is not zeroed with ECC and persistence disabled. This indicates that memory zeroing in the ECC case is not implemented for security reasons but only to properly support ECC mode.

In the case of a soft reboot of the machine or a reset of the GPU, the driver is unloaded and reloaded independently of the persistence mode. There is no information leakage between *taint* and *search* with ECC enabled in these cases.

5.5.2 Virtualized environment

Within a guest virtual machine, we observed information leakage when switching user between *taint* and *search*, which is the same behavior as in a native environment. The soft reboot and the GPU reset are also giving different results depending on ECC, showing information leakage when ECC is disabled, and no leakage when ECC is enabled. Consistently with the native environment, there was no information leakage after a hard reboot. Information leakage on these setups threatens the confidentiality between users and applications of the same guest virtual machine.

To investigate the role of the hypervisor, we are interested in knowing whether a guest virtual machine can retrieve data in the GPU memory left by a previous guest. For that matter, we create a guest virtual machine running NVIDIA driver on Ubuntu, launch the *taint* program and then destroy the virtual machine. Afterwards, we create another guest virtual machine and launch the *search* program. We retrieved data on both Xen and KVM, revealing that information has leaked. This result indicates a clear violation of the isolation that the hypervisor must maintain between two guest virtual machines.

5.5.3 Cloud environment

Within a guest virtual machine (called *instance* on Amazon EC2), we obtain the same results as in the virtualized environment. Information leakage occurs with ECC disabled when there is a user switch, after a soft reboot of the virtual machine or a reset of the GPU.

In the default configuration of Amazon GPU instances, ECC is enabled and persistence is disabled. In accordance with our previous experiments, it means that GPU memory is cleaned, and it is supposed to prevent a user from accessing the memory of previous users. However, a user that deactivates ECC to have more memory available (or uses a virtual machine image configured this way) may not be protected. Based on our observations, we imagine a scenario where an attacker rents many instances and disables ECC – or provides a custom image that disables ECC to numerous victims. Slaviero et al. [SMA09] showed that it is possible to pollute the Amazon Machine Image market with virtual machine images prepared by an attacker. The attacker then waits for its victim to launch an instance where the ECC has been disabled. When the victim releases the instance, the attacker tries to launch his own instance on the same physical machine. While this is difficult, several studies [RTSS09, VZRS15, XWW15] showed that it is possible to exploit and influence virtual machine placement in Amazon. The attacker then runs the *search* program to seek data in the GPU memory. We did not implement this attack as we would have needed to rent numerous instances, without any guarantee to retrieve the same physical machine after freeing one.

We therefore contacted Amazon security team, who mentioned that they were already addressing such concerns in their pre-provisioning workflow, *i.e.*, before allocating a new instance to a user. However, without further details on how GPU memory is cleaned, there is no guarantee that Amazon performs this correctly. In addition to this, in absence of formal industry recommendations, we cannot exclude the existence of data leakage in other GPU cloud providers.

5.6 Accessing memory through PCI configuration space

The access method that leverages GPGPU runtime has the disadvantage of showing a partial view of the GPU memory, *i.e.*, only what can be accessed via the GPU MMU. In this section, we show a method to access the GPU memory through the PCI configuration space, in a driver agnostic way.

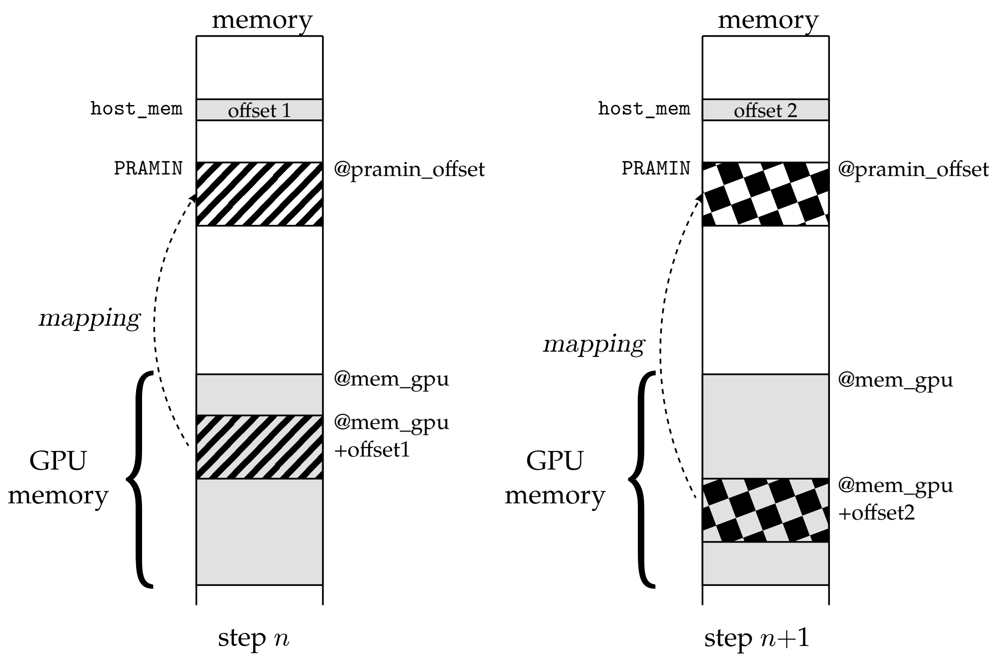

Figure 5.1: Accessing GPU memory via PCI configuration space: PRAMIN mapping is used to access 1MB of the GPU physical memory, at address configured in the register `host_mem`. We depict two consecutive steps in Algorithm 4 `while` loop.

5.6.1 Native environment

There are two methods to perform I/O operations between the CPU and I/O devices: Memory-Mapped I/O (MMIO) and Port-mapped I/O (PIO). The mapping of the device memory to the MMIO or PIO address space is configured in the Base Address Registers (BAR), in the PCI configuration space. The PCI configuration space is a set of registers that allow the configuration of PCI devices. Reads and writes can be initiated by the legacy x86 I/O address space, and memory-mapped I/O.

For NVIDIA GPUs, the BARs were obtained by a reverse-engineering work of the open-source community. BAR0 contains MMIO registers, documented in the Envytools git [Env]. The registers are architecture dependent, but the area we are interested in remains the same for the architectures Tesla, Fermi and Kepler. The mapping at `0x700000-0x7fffff`, called PRAMIN, can be used

Algorithm 4 Accessing memory through PRAMIN

```
pramin_offset ← 0x700000
host_mem ← 0x0
vram[size]
while i < size do
    read(pramin_offset, vram[i], 0x100000)
    host_mem ← host_mem + 0x100000
end while
```

to access any part of video memory by its physical address. It is used as a 1MB window to physical memory, and its base address can be set using the register `HOST_MEM` at the address `0x1700`. Figure 5.1 illustrates this access.

The access to video RAM is done through the following steps. First, `HOST_MEM` is set to `0x0` and we read the PRAMIN BAR (1MB) – this way we are able to read the first 1MB of the GPU physical memory. We then add 1MB to `HOST_MEM` and re-read PRAMIN. This step is done again until the whole memory has been accessed. Algorithm 4 summarizes these steps. We use the read and write functions of Envytools [Env] (`nva_wr32` and `nva_rd8`), that in turn use `libpciaccess` to access the PCI configuration space.

Consistently with the experiments leveraging a GPGPU runtime, we observe information leakage after a soft reboot and a reset of the GPU. There is no information leakage after a hard reboot. Changing user does not apply in this setup since we need to be root to access the PCI configuration space.

Accessing memory through PCI configuration space gives a complete snapshot of the GPU memory and bypasses the GPU MMU. The advantage of such method is that it is capable of bypassing some memory cleanup measures implemented at the applicative level. We discuss this aspect in Section 5.7.

5.6.2 Virtualized and cloud environment

Xen provides I/O virtualization by means of emulation for its HVM guests with the QEMU device model (QEMU-dm) daemon that runs in Dom0. When a guest is configured with a device in direct device assignment mode, QEMU-dm reads its PCI configuration space register, and then replicates it in a virtual PCI configuration space. QEMU-dm maps MMIO and PIO into the guest memory space, and configures the IOMMU to grant the guest OS access to these memory regions. However, QEMU-dm emulates some configuration space registers like BAR for security reasons, so that an attacker cannot change the memory mapping of the device to another device attached to another virtual machine, or to the hypervisor. Other registers like the command register are not emulated.

Our access method leverages BAR registers to access the GPU memory. We tested this method on our Xen setup and obtained garbage (series of `0xffff` values), confirming that the access to the registers is emulated, which prevented us from effectively accessing the memory. The results are the same for Amazon GPU instances. These setups are then showing no information leakage. To circumvent the protection of BAR registers, an attacker may try to attack the virtualization mechanisms themselves.

5.7 Countermeasures

We divide the possible countermeasures in three categories: changes in existing runtimes, steps that can be taken by cloud providers, and those that can already be initiated by a user using only calls to existing APIs.

5.7.1 GPGPU runtimes

Di Pietro et al. [DLV13] suggested an approach to be implemented in runtimes. Their solution is to zero-fill buffers at allocation time, as it is done when an operating system allocates a new physical page of memory to a process. This solution targets an attacker that uses GPGPU runtime to launch his attack, however, it does not protect from an attacker who accesses memory through PCI configuration space, since he will not allocate memory. In this case, it would be better to clear memory at deallocation time. In both cases, zero-filling buffers entails performance issues as the memory bandwidth is generally a bottleneck for GPGPU applications. Di Pietro et al. assessed the impact of the `cudaMemset` function that is used for zeroing buffers. The overhead turns out to be linearly proportional to the buffer size.

5.7.2 Hypervisors and cloud providers

Cloud providers can already take measures to protect their customers. The necessary steps before handing an instance to a customer include cleanup of the GPU memory. This is the approach that appears to be taken by Amazon, which seems to implement proper memory cleaning. This approach should also be implemented in hypervisors. Although this would not eliminate attacks within a virtual machine, a pre-provisioning approach effectively eliminates attacks across virtual machines with no overhead for customers.

5.7.3 Defensive programming

In the absence of the two types of countermeasures above, a security-conscious programmer that writes his own programs and accepts a performance penalty can clear the buffer before freeing memory with a function such as `cudaMemset`. If the end-user can not modify the program, he should erase the GPU memory when finishing an execution on a GPU. This countermeasure seems trivial, nevertheless its practical implementation can be difficult due to the complicated memory hierarchy present in GPUs (e.g., access mechanisms depend on the type of memory). A standalone CUDA program that cleans the memory would allocate the maximum amount of memory, and then overwrite it (e.g., with zeros). However, this solution relies on the CUDA memory manager, which does not guarantee the allocation of the whole memory. Portions of memory risk not to be properly erased because of fragmentation issues. We built an experiment to illustrate this. We run a first CUDA program for some time, then we stop it to run a second CUDA program that cleans the memory. We finally dump the memory via PRAMIN to access the whole memory. We recovered a portion of the memory that was not cleaned by the second CUDA program, demonstrating clear limitations of this countermeasure.

A practical solution for NVIDIA Tesla GPUs that benefit from ECC memory is to enable ECC and reload the driver, or to reset the GPU when ECC is enabled. As we saw in our experiments Section 5.5.1, these sequences of actions clear the memory.

5.8 Related work

Using the CUDA framework, Di Pietro et al. [DLV13] showed that GPU architectures are vulnerable to information leakage, mainly due to memory isolation issues. The leakage affects different memory spaces in GPU: global memory, shared memory, and registers. Di Pietro et al. also showed that current implementations of AES cipher that leverage GPUs allow recovering both plaintext and encryption key in GPU global memory. Bress et al. [BKS13] considered using these vulnerabilities to perform forensic investigations. Nevertheless, they noted that we cannot guarantee that calls to the CUDA API do not modify the memory. These two articles began to pave the way of GPU security. However, they did not evaluate information leakage by GPUs in the context of virtualization that is characteristic of cloud computing. In contrast, we analyzed several approaches to access the GPU memory, experimenting with different drivers and GPGPU runtimes, as well as the PCI configuration space. We also extended these findings to virtualized environments.

Subsequent work of Lee et al. [LKKK14] showed further the applicability of this type of information leakage by recovering textures rendered by the GPU for web browsers. This attack allows inferring which web pages are visited by a victim. Danisevskis et al. [DPS13] showed that mobile GPUs, which are becoming increasingly more common, and in particular their DMA capabilities, can also be abused for malware delivery.

5.9 Conclusions and perspectives

In this chapter, we evaluated the confidentiality issues that are caused by the recent advent of GPU virtualization. Our experiments in native and virtualized environments showed that the driver, operating system, hypervisor and the GPU card itself do not implement any security related memory cleanup measure. As a result, we observed information leakage from one user to another, and in particular from one virtual machine to another in a virtualized environment. Amazon EC2 seems to implement proper GPU memory cleaning at the provisioning of an instance; we could thus not confirm any information leakage from one Amazon instance to another. However, because of the general lack of GPU memory zeroing, we cannot generally exclude the existence of data leakage in cloud computing environments.

The rise of GPGPU increases the attack surface and urges programmers and industry to handle GPU memory with the same care as main memory. For this matter, industry should include GPU memory cleaning in its best practices. We provided a set of recommendations for proper memory cleanup at the various layers involved in GPU virtualization (application, driver, hypervisor).

The most recent trend in GPU virtualization is the shift from sequential sharing of a GPU card to simultaneous sharing between several tenants. Indeed, NVIDIA launched a new series of GPUs called GRID that handle virtualization in hardware with a dedicated hypervisor. In this context, memory isolation is even more challenging, and new issues arise such as covert and side channels. This effectively allows the other attacker model we described in Section 5.2, *i.e.*, the *parallel attacker*. In context of native environments, integrated GPUs are another unexplored research topic. Being manufactured on the same die as the CPU, they must fit on a smaller surface than a discrete GPU and are thus less efficient. Moreover, they do not have any dedicated memory and therefore they use the system memory. Yet, the latest range of integrated GPUs increased in performance, and they are now much more widespread. The consequences of this integration to the CPU in terms of security will be explored in a future work.

6 Conclusions and future directions

Contents

Information leakage due to shared hardware has recently gained a new impetus with the appearance of cloud computing environments. Indeed, these environments heavily use virtualization to co-locate several virtual machines, that can be owned by different customers, on the same physical machine. This thesis focuses on attacks due to memory isolation issues as well as covert and side channels performed at the microarchitectural level. Indeed, main memory and some microarchitectural elements are shared between virtual machines and accessible without physical access. However, these attacks are highly dependent on the hardware, which evolves constantly. Our work was thus motivated by the following question: *How do the recent evolutions impact information leakage due to hardware sharing?*

6.1 Contributions

In this thesis, we explored the impact of evolutions in recent hardware in terms of information leakage on shared hardware and applied our findings to virtualized environments.

Exploiting hardware features to build a low-requirement covert channel Cross-core cache attacks target the last-level cache that is shared by all CPU cores. Previous attacks on the last-level cache relied on shared memory in order to evict cache lines. However, disabling shared memory is sufficient to circumvent these attacks, and this countermeasure is already in production in some cloud environments. We demonstrated a cross-core covert channel without using any shared memory, exploiting the shared and inclusive properties of the last-level cache. We evaluated this covert channel in native and virtualized environments.

Reverse-engineering cache internals to improve attacks The function that maps a physical address to a last-level cache slice is undocumented on Intel CPUs. This renders attacks either slow, difficult or even impossible to perform, because of the loss of precision in the cache collisions. We built an automatic and generic method for reverse-engineering the complex addressing function. We evaluated our method and retrieved the function for a wide range of processors, encompassing different microarchitectures and different numbers of cores. We demonstrated the broad applicability of these findings by accelerating our cache covert channel and adapting Rowhammer, a DRAM vulnerability, in sandboxed JavaScript.

Investigating setups causing information leakage on GPU memory GPUs have been designed to provide maximum performance. They have not been designed with security in mind, nor to support concurrent access. Yet, they are increasingly used in cloud computing setups. We documented the security implications of the different GPU virtualization techniques. We systematically experimented and analyzed the behavior of GPU global memory in the case of direct device assignment. We found information leakage in scenarios such as an attacker launching a virtual machine after a victim's virtual machine using the same GPU. With these results, we clearly underlined bypasses of the isolation mechanisms of virtualization. We proposed countermeasures to be implemented by cloud providers and end users.

6.2 Future directions

Hardware is in constant evolution, requiring a continuous effort from both attacks and defenses to accommodate to these changes. The subjects explored in this thesis and our results open new perspectives in this domain and can be further developed in the following directions.

6.2.1 Attack techniques

Exploiting shared caches in new attacks A shared cache is one of the conditions that render cross-core attacks possible, as we showed in Chapter 3. We envision three more issues in this direction. First, in some processors the cache is not only shared by the CPU cores, but also with the integrated GPU, when available. With advances in GPU malware, it will be necessary to investigate the possibility of covert communication between the CPU and the integrated GPU through this shared cache. Second, in some Haswell and Broadwell CPUs, another level of cache can be found: the L4 cache. It is an eDRAM die that can be used by the integrated GPU and the CPU, and is also shared across cores. The CPU uses it as a *victim cache*, *i.e.*, a line evicted from L3 will go to L4 before being evicted from the whole hierarchy. This change in cache design creates opportunities to reevaluate current cache attacks by taking into account this new level, and to investigate possibilities of new cache attacks. Third, an interesting direction would be to examine non-x86 systems, e.g., ARM. Indeed, ARM processors are now massively used in ubiquitous devices such as smartphones. A side channel has already been demonstrated on the private L1 cache [SP13], however not yet on the shared L2. As this cache level is exclusive for data, current attack techniques are not directly applicable, and exploiting exclusive caches remains a challenge.

New GPU architectures Subsequently to our study on information leakage on GPU memory in Chapter 5, NVIDIA launched new GPUs that support virtualization in hardware, now adopted by Amazon EC2. As these GPUs support concurrent users, we expect that indirect information leakage such as covert and side channels will be possible.

6.2.2 Defense techniques

Our work on cache attacks in Chapter 3 and 4 demonstrates that current countermeasures that disable shared memory in virtualized environments and the `clflush` instruction in sandboxes are not sufficient. Indeed, the root causes of interferences between cores do not lie in the shared memory or the `clflush` instruction, but rather in the inclusive and shared cache. As these two features are key elements in today's cache performance, they are likely to stay. Adequate countermeasures must thus take them into account. Proposed countermeasures that are focused on the L1 cache are not applicable to the last-level cache, for performance reasons. For instance, countermeasures that flush the cache periodically to induce noise would in practice flush all the cache hierarchy if applied to the last-level cache, due to its inclusive property. Moreover, the recent advances in cache attacks are led by advances in performance. Finding efficient countermeasures that have a low performance impact is thus a challenge, however it is a necessity for their adoption.

Our work on the reverse engineering of the last-level cache addressing function in Chapter 4 opens a perspective for a more fine-grained page coloring system at the software level. Indeed, having the addressing function allows a defense mechanism to target and isolate sets from different programs or virtual machines more precisely, thus limiting the impact on performance.

Additionally, our usage of hardware performance counters in Chapter 4 can be extended to detect cache attacks in order to later stop them without causing a loss of performance to the whole system. Indeed, cache attacks incur a significant number of cache misses, that can be monitored by performance counters. A thorough evaluation needs to be conducted to measure the efficiency of this system.

6.2.3 Expanding knowledge of CPU internals

For both attacks and defense, it is necessary to understand in detail the internals of increasingly complex CPUs. Their usage is extensively documented by vendors, however some performance-critical parts like addressing functions and replacement policies remain undocumented, seemingly to keep some advantage in the performance race.

Following the reverse-engineering work in Chapter 4, we foresee another direct application. The Ivy Bridge microarchitecture introduced another undocumented addressing function, that maps a physical address to a DRAM channel. Additionally, Xeon processors can monitor events associated to each DRAM channel. Using the same technique as with the events associated to each slice would retrieve the addressing function of DRAM channels.

More generally, hardware performance counters can monitor plenty of hardware events, and can be of a great help to understand the series of events happening at the hardware level. Indeed, a useful tool would be an accurate CPU simulator. CPU simulators are currently used by the research community interested in hardware performance, to benchmark and compare their respective contributions. However, these simulators currently also suffer from the lack of documentation by vendors. They are thus not reliable when looking at some components in detail, e.g., for a security analysis that needs to be accurate relatively to real-world hardware. Continuing this reverse-engineering effort is helpful for the research community at large, beyond the security community.

A

Résumé en français

A.1 Introduction

A.1.1 Contexte

Les environnements *cloud* ont été introduits lors de cette dernière décennie et ont gagné en popularité depuis. Ils apportent aux clients, particuliers et entreprises, des solutions pour le calcul et le stockage dans des centres dédiés. Pour les clients, le bénéfice est la simplicité : les mêmes services s'exécutent sur différentes plateformes physiques, sans avoir à considérer les spécificités du matériel. Cela décharge également les besoins de gestion de l'infrastructure. Pour les fournisseurs de service, le bénéfice est le rapport coût-efficacité: plusieurs machines virtuelles, qui peuvent être possédées par différents clients, tournent sur la même machine physique. Le cloud se base énormément sur la virtualisation, qui consiste en le découplage des services logiciels par rapport au matériel. Le partage matériel est un aspect central des environnements cloud. Entre autres, deux importantes pièces matérielles sont partagées aujourd'hui : le CPU et le GPU. Le matériel partagé entre différents clients cause des menaces de fuites d'information. En particulier, le CPU est accédé de manière concurrente par différents utilisateurs, et la mémoire cache des CPUs est énormément partagée. Cela mène à des canaux cachés (*covert channels*) et des canaux auxiliaires (*side channels*). Le GPU est partagé sur le temps, c'est à dire que deux utilisateurs ne peuvent pas l'utiliser au même moment, mais ils peuvent l'utiliser les uns après les autres. L'isolation mémoire est cruciale dans ce cas pour prévenir les fuites d'information. Les attaques sur ces systèmes visent soient à activement échanger des données secrètes d'un processus à un autre, ou à espionner un processus pour exfiltrer des données secrètes d'une victime.

A.1.2 Problématique

Les fuites d'informations dues au matériel partagé sont un sujet connu qui a été largement étudié. Cependant, ces attaques sont très dépendantes du matériel. Nous voyons également des évolutions aussi bien dans la construction du matériel que dans son adoption. Premièrement, la microarchitecture des CPUs change fréquemment. Par exemple, Intel en fabrique une nouvelle quasiment tous les ans depuis 2009. Deuxièmement, les GPUs ont été construits dans le but de fournir un maximum de performance et non pour des accès concurrents, et sans prendre en compte des critères de sécurité. Néanmoins, ils ont récemment été proposés par les fournisseurs de service cloud. En raison de l'évolution du matériel ou des contre-mesures en production, certaines attaques sont rendues plus complexes voire impossible à réaliser. En revanche, certaines modifications sont réalisées avec pour seul but la performance, ce qui est souvent en contradiction avec la sécurité. Nous posons donc la question suivante : *Comment ces récentes évolutions impactent les fuites d'informations dues au partage matériel ?*

Cette question donne lieu à plusieurs défis en ce qui concerne l'investigation de ces questions de sécurité. En effet, nous sommes face à deux niveaux d'obscurité. Le premier est celui du fournisseur de services cloud. Pour des raisons aussi diverses que les préoccupations de sécurité ou le modèle économique, les fournisseurs de services cloud sont réticents à donner des détails sur leur infrastructure. Le second est le matériel en lui-même. Étant de plus en plus complexe, les CPUs comme les GPUs sont construits avec certaines parties cruciales en terme de performance, mais non documentées.

A.1.3 Contributions

Le but de cette thèse est d'étudier l'impact des évolutions du matériel récent en terme de fuites d'informations sur le matériel partagé. Nous appliquons également nos découvertes aux environnements virtualisés qui sont largement utilisés aujourd'hui et constituent un cas d'utilisation naturel de matériel partagé. Cette thèse présente les travaux réalisés durant ma thèse et apporte des contributions selon trois axes principaux.

Canaux cachés au niveau des mémoires caches des processeurs Les canaux cachés ont démontré pouvoir transgresser l'isolation des environnements virtualisés, et typiquement, de permettre l'exfiltration de données. Plusieurs canaux cachés se basant sur la mémoire cache des processeurs ont été proposés. Cependant, ces canaux cachés sont soit lents ou inapplicables en raison de l'incertitude sur l'adressage du cache. Cette incertitude est causée par le niveau supplémentaire d'indirection dans les environnements virtualisés, ainsi que par le mode d'adressage du dernier niveau de cache dans les processeurs récents. Utiliser de la mémoire partagée (comme la dé-duplication mémoire entre deux machines virtuelles) permet de résoudre cette incertitude, mais la mémoire partagée n'est pas toujours disponible en pratique dans le cloud. En effet, la dé-duplication est désactivée par la plupart des fournisseurs de service cloud, comme Amazon Web Services. Nous avons construit C5, un canal caché qui s'attaque à la problématique de l'incertitude sur l'adressage sans requérir de mémoire partagée, en faisant un canal caché rapide et pratique. Ce canal caché peut transférer des messages entre différents coeurs d'un processeur récent. Il cible le dernier niveau de cache qui est partagé entre tous les coeurs, et exploite la caractéristique d'inclusivité de ce niveau de cache, qui permet à un coeur de supprimer des lignes du premier niveau de cache privé d'un autre coeur. Nous avons évalué notre canal caché en environnement natif et virtualisé. En particulier, nous avons établi un canal caché entre machines virtuelles tournant sur des coeurs différents. Nous avons mesuré une vitesse d'un ordre de grandeur supérieur aux canaux cachés de la littérature dans les mêmes conditions.

Rétro-ingénierie de la fonction d'adressage du dernier niveau de cache dans les processeurs Intel Le dernier niveau de cache des processeurs Intel est partagé en *slices*. Prédire la slice utilisée par une adresse mémoire est simple dans les processeurs plus anciens, mais les processeurs récents utilisent une fonction d'adressage non documentée. Cela rend certaines attaques plus difficiles, et d'autres totalement impossibles, en raison du manque de précision dans la prédiction des collisions dans le cache. L'état de l'art avait seulement retrouvé la fonction manuellement et pour un seul modèle de processeur. Nous avons construit une méthode automatique et générique pour effectuer la rétro-ingénierie de cette fonction d'adressage, rendant par conséquent la classe des attaques sur les caches grandement pratique. Notre méthode s'appuie sur les compteurs de performance matériels des processeurs pour déterminer la slice du cache à laquelle une adresse mémoire est mappée. Nous avons montré que notre méthode donne une description plus précise de la fonction d'adressage que les travaux existants. Nous avons validé notre méthode en effectuant la rétro-ingénierie de cette fonction sur un ensemble varié de processeurs Intel, comprenant les microarchitectures Sandy Bride, Ivy Bridge et Haswell, avec différents nombre de coeurs, et pour les gammes de processeurs mobiles et de serveurs. Nous avons montré que connaître la fonction d'adressage du dernier niveau de cache permet d'améliorer C5 de plusieurs ordres de grandeur.

Fuites d'information dans les GPUs virtualisés Peu d'études ont été conduites sur les implications en terme de sécurité du calcul générique sur un processeur graphique (GPGPU) combiné aux environnements cloud. Notre objectif a été de souligner les fuites d'information dues aux GPUs dans les environnements virtualisés et de cloud. Nous avons étudié les différentes techniques de virtualisation des GPUs, ainsi que leurs implications en terme de sécurité. Nous avons analysé de manière systématique le comportement de la mémoire globale des GPUs dans le cas du *direct device assignment*. Nous avons trouvé que la mémoire globale des GPUs n'est correctement nettoyée que dans certaines configurations, seulement en effet secondaire des codes correcteurs d'erreurs (ECC), et non pas pour des raisons de sécurité. Ainsi, un attaquant peut retrouver des données d'applications GPGPU précédemment exécutées dans une variété de scénarios. Ces scénarios incluent des situations dans lesquelles un attaquant lance une machine virtuelle après celle d'une victime ayant utilisé le même GPU. Cette attaque transgresse clairement l'isolation qu'un hyperviseur est censé apporter aux environnements virtualisés. Le nettoyage de la mémoire n'est pas implémenté par le GPU lui-même, et nous ne pouvons pas exclure de manière définitive la possibilité de fuites d'information dans les environnements cloud. Nous avons en plus discuté de contre-mesures possibles pour les utilisateurs et les fournisseurs de services cloud.

A.1.4 Organisation de la thèse

Cette thèse est organisée comme suit.

Le chapitre 2 passe en revue l'état de l'art ainsi que la documentation technique nécessaire à la lecture de cette thèse. Il couvre tout d'abord l'architecture et la virtualisation des systèmes x86. Il inclut également des détails sur les fuites d'informations sur les ressources partagées comme le bus mémoire et le CPU, avec un intérêt particulier pour les caches des CPUs et la mémoire des GPUs.

Le chapitre 3 présente C5, un nouveau canal caché sur le dernier niveau de cache inclusif. Ce canal caché prend en compte l'évolution des CPUs récents. Nous avons évalué sa vitesse et son taux d'erreur sur différentes expériences, notamment en passant des messages d'un coeur à l'autre et entre machines virtuelles.

Le chapitre 4 détaille comme nous avons effectué la rétro-ingénierie de la fonction d'adressage du dernier niveau de cache dans les processeurs Intel. Nous avons évalué notre approche en retrouvant la fonction sur un large ensemble de processeurs différents. Nous montrons également des applications en terme de sécurité qui découlent de ces travaux.

Le chapitre 5 documente l'impact des techniques de virtualisation des GPUs en termes de sécurité. Nous investiguons de manière systématique les fuites d'informations sur la mémoire des GPUs, en particulier dans les environnements virtualisés. Nous détaillons deux méthodes pour accéder à la mémoire d'un GPU, qui requièrent différents niveaux de privilèges.

Le chapitre 6 conclut et donne les perspectives sur les travaux futurs.

A.2 Contourner l'adressage complexe : le canal caché C5

Les canaux cachés sont utilisés pour exfiltrer des informations sensibles, et peuvent aussi être utilisés comme tests de co-résidence dans le cloud [ZJOR11]. Il y a plusieurs défis pour réaliser des canaux cachés entre machines virtuelles. Premièrement, la migration des programmes entre les différents coeurs réduit drastiquement la vitesse des canaux cachés qui ne marchent pas entre différents coeurs [XBJ$^+$11]. Deuxièmement, l'exécution simultanée de machines virtuelles entre différents coeurs empêche un ordonnancement de type round-robin strict entre un émetteur et un récepteur [WXW12]. Troisièmement, la traduction d'adresse virtuelle vers physique ainsi que les fonctions qui mappent une adresse vers un set de cache ne sont pas exposées aux processus, et induisent ainsi une incertitude sur l'emplacement des données dans le cache. Cette incertitude sur l'adressage (appelée *addressing uncertainty* dans [WXW12]) empêche un émetteur et un récepteur de parvenir à un accord sur un emplacement sur lequel travailler.

Les canaux cachés qui ne prennent pas en compte cette incertitude sur l'adressage sont limités à utiliser le premier niveau de cache privé sur les processeurs modernes. Cela réduit dramatiquement la vitesse en environnement virtualisé. Une méthode pour circonvenir à ce problème est de dépendre de la dé-duplication mémoire offerte par l'hyperviseur ou le système d'exploitation. Avec la dé-duplication, les pages mémoires en commun de deux processeurs utilisent les mêmes lignes de cache. Cependant, la dé-duplication est désactivée par certains fournisseurs de services cloud [BRPG15], ce qui rend cette attaque inexploitable dans certaines situations.

Dans ce chapitre, nous présentons un nouveau canal caché, appelé C5. Nous différencions notre canal caché de l'état de l'art en prenant en compte le problème de l'incertitude sur l'adressage sans dépendre de la mémoire partagée. Notre canal caché marche entre deux machines virtuelles qui tournent sur n'importe quel coeur d'un même processeur. Il utilise le fait que le dernier niveau de cache est à la fois partagé et est inclusif des niveaux L1 et L2 dans les processeurs modernes. Nous obtenons les hautes vitesses de 1291bps dans un environnement natif et 751bps dans un environnement virtualisé, soutenant ce canal caché comme une attaque pratique.

A.3 Rétro-ingénierie de l'adressage du dernier niveau de cache

Les attaques sur les caches peuvent opérer à tous les niveaux de cache : L1, L2 ou dernier niveau de cache. Les attaques sur les niveaux L1 et L2 restreignent l'attaquant à être sur le même coeur que sa victime. Cette hypothèse est trop forte sur un processeur multi-coeurs quand l'attaquant et la victime migrent de coeurs [RTSS09, XBJ$^+$11]. Nous nous concentrons donc sur les attaques sur le dernier niveau de cache, qui est partagé entre les coeurs dans les processeurs modernes. Les attaques sur le dernier niveau de cache sont plus puissantes, car l'attaquant et la victime peuvent ainsi être sur des coeurs différents, mais elles sont également plus difficiles. Sans utiliser de mémoire partagée, un attaquant doit trouver des adresses qui mappent un même set de cache, et exploiter la politique de remplacement de cache pour expulser les lignes du set.

L'attaquant est face à deux problèmes : le dernier niveau de cache est adressé physiquement, et les processeurs modernes mappent les adresses physiques aux slices de cache en utilisant une fonction appelée adressage complexe, qui n'est pas documentée. Nous avons détaillé dans le dernier chapitre une méthode pour contourner cette fonction complexe, en expulsant les lignes de tout le dernier niveau de cache. C'est un moyen pratique de réaliser des canaux cachés comme nous l'avons montré, mais ce n'est en revanche pas une méthode qui est assez fine pour exécuter, par exemple, des canaux auxiliaires sur des algorithmes cryptographiques. En outre, les approches précédentes ont déjà effectué la rétro-ingénierie de cette fonction, manuellement et pour un processeur Sandy Bridge spécifique [HWH13, Sea15b]. Ils ont cependant utilisé une attaque temporelle qui n'est pas capable de retrouver tous les bits de la fonction, menant à une fonction partielle seulement. La fonction d'autres processeurs est inconnue.

Dans ce chapitre, nous présentons une méthode nouvelle et automatique pour effectuer la rétro-ingénierie de la fonction d'adressage du dernier niveau de cache. Nous utilisons les compteurs de performance matériels pour contrôler les événements associés aux slices de cache, dans le but de mapper les adresses physiques à chaque slice. Nous retrouvons donc tous les bits de la fonction. Nous avons évalué notre méthode sur une grande variété de processeurs, comprenant les microarchitectures Sandy Bridge, Ivy Bridge et Haswell, pour différents nombres de coeurs.

A.4 Fuites d'informations sur la mémoire des GPUs

Les GPUs bénéficient d'un grand intérêt de la part de la communauté scientifique. En effet, le calcul générique sur GPU permet d'effectuer des calculs génériques massivement parallèles en utilisant la parallélisme intrinsèque des GPUs. Les applications sont diverses comme la finance [BSS10, KP05, TB10], le chiffrement [gE, Yam07], et le mining de Bitcoins [Kol]. Les GPUs ont récemment été proposés par plusieurs fournisseurs de services cloud pour fournir un paiement à l'utilisation et à la demande d'un matériel autrement très coûteux. En outre, les applications utilisant la virtualisation pour du rendu graphique sont émergentes. Les appareils peu performants, comme les tablettes, peuvent être utilisés pour des jeux vidéo sur des plate-formes de *cloud gaming*. Des ordinateurs portables ou stations de travail ordinaires peuvent exécuter des taches comme du montage vidéo ou de conception assistée par ordinateur sur des systèmes virtualisés.

Les GPUs ont été conçus pour fournir un maximum de performance. Cependant, ils n'ont pas été conçus pour des accès concurrents, c'est à dire pour supporter de la virtualisation ou plusieurs utilisateurs simultanément. Il est connu que les buffers des GPUs ne sont pas nettoyés quand ils sont alloués [Ker12]. Cela pose des questions de confidentialité entre différents programmes ou différents utilisateurs quand les GPUs sont utilisés en environnement natif sur des ordinateurs personnels [DLV13]. Clairement, la surface d'attaque est plus grande dans un environnement cloud quand plusieurs utilisateurs exploitent le même GPU les uns après les autres, ou même simultanément. Cependant, cette problématique n'a jamais été étudiée. De plus, identifier les fuites d'informations possibles dans ces environnements est un problème difficile dû au deux couches d'obscurité : le fournisseur de services cloud ainsi que le GPU lui-même.

Dans ce chapitre, nous étudions les fuites d'informations sur les GPUs et évaluons ses impacts possibles sur les clouds GPUs. Nous expérimentons et analysons de manière systématique le comportement de la mémoire globale du GPU en environnements natif et virtualisé. Nous différenciant de l'état de l'art [BKS13, DLV13], nous montrons comment un attaquant peut retrouver des informations de la mémoire globale du GPU en utilisant une variété de drivers et de frameworks. Par ailleurs, nous trouvons que dans les rares cas où la mémoire globale du GPU est nettoyée, elle l'est seulement à cause d'un effet de bord de la correction d'erreurs (ECC), et non pour des raisons de sécurité. Nous proposons également une méthode pour retrouver la mémoire du GPU d'une façon agnostique au driver, qui contourne des mesures de nettoyage de la mémoire qu'un programmeur consciencieux pourrait implémenter.

A.5 Travaux futurs

Dans cette thèse, nous avons exploré l'impact des évolutions du matériel récent en termes de fuites d'information sur du matériel partagé, et avons appliqué nos découvertes aux environnements virtualisés.

Le matériel est en constante évolution, ce qui requiert un effort continu de la part à la fois des attaquants et des défenseurs, pour s'accommoder de ces changements. Les sujets explorés dans cette thèse ouvrent de nouvelles perspectives dans ce domaine et peuvent être développés dans les directions suivantes.

A.5.1 Nouvelles attaques

Les processeurs continuent d'évoluer, ainsi que les caractéristiques de leurs caches. De nouvelles attaques sont donc envisageables, en tirant parti du cache partagé entre le CPU et le GPU intégré dans certains modèles de processeurs Intel, ou par exemple en regardant les implications du nouveau niveau de cache dans les tous dernières microarchitectures. Une autre direction intéressante est de regarder l'architecture ARM, omniprésente par exemple dans les smartphones, et qui n'a pour l'instant pas reçu la même attention que l'architecture x86. Les GPUs continuent également d'évoluer, et de nouvelles architectures supportent maintenant la virtualisation au niveau hardware. Si l'isolation mémoire a des chances d'avoir été au coeur du design de ces GPUs, il peut subsister, comme au niveau des CPUs, des possibilités de fuites d'information par des canaux cachés ou des canaux auxiliaires.

A.5.2 Contre-mesures

Ces travaux de thèse ont montré à quel point les techniques défensives sont plus que jamais importantes face aux attaques sur les caches, et que les contre-mesures logicielles appliquées jusque ici (comme la désactivation de la déduplication mémoire) ne sont pas efficaces contre ces nouvelles attaques. De nombreuses solutions existent dans la littérature au niveau matériel mais ces solutions ne sont applicables que par les constructeurs et, comme elles induisent des pertes de performances, elles ne sont pas introduites dans le design des nouveaux processeurs. Des contre-mesures logicielles efficaces et minimisant les pertes de performance sont donc indispensables. Nos travaux sur la rétro-ingénierie de la fonction d'adressage du cache ouvrent des perspectives quand à un partitionnement à grain fin du cache et donc entraînant peu de pertes de performance. Il est également envisageable d'utiliser les compteurs de performance pour monitorer le système et détecter des attaques sur les caches.

A.5.3 Élargir les connaissances du fonctionnement interne des CPUs

Que ce soit pour les attaques ou les contre-mesures, il est nécessaire de comprendre en détail le fonctionnement interne des CPUs, qui sont de plus en plus complexes. Si leur usage est largement documenté par les vendeurs, certaines parties critiques pour les performances comme les fonctions d'adressage et les politiques de remplacement de cache restent non documentées, certainement afin de préserver un avantage dans la course à la performance.

En continuant le travail de rétro-ingénierie du chapitre 4, nous voyons une autre application directe. La microarchitecture Ivy Bridge a introduit une nouvelle fonction d'adressage non documentée, qui mappe une adresse physique à un canal de DRAM. De plus, les processeurs Xeon peuvent monitorer les événements associés à chaque canal de DRAM. En utilisant la même technique que pour les slices de cache, il serait possible de retrouver la fonction d'adressage des canaux de DRAM. Plus généralement, les compteurs de performance matériels peuvent monitorer beaucoup de types d'événements matériels, et peuvent être d'une grande aide pour comprendre les séries d'événements qui surviennent au niveau matériel.

Le travail de rétro-ingénierie est également bénéfique à la communauté scientifique au sens large, au delà de la communauté de sécurité, pour tous les travaux de recherche qui ont trait à la performance et qui ont besoin de modèles fiables et le plus proche possible du matériel réel.

B

Accurate timing measurements

For accurate timing measurements taking into account out-of-order execution, we use the following method, described in more detail in [Int10].

```
#define begin_measurement(begin_high, begin_low) \
  asm volatile ("CPUID\n\t" \
    "RDTSC\n\t" \
    "mov %%edx, %0\n\t" \
    "mov %%eax, %1\n\t" \
    : "=r" (begin_high), "=r" (begin_low) \
    : \
    : "%rax", "%rbx", "%rcx", "%rdx");


#define end_measurement(end_high, end_low) \
  asm volatile("RDTSCP\n\t" \
    "mov %%edx, %0\n\t" \
    "mov %%eax, %1\n\t" \
    "CPUID\n\t" \
    : "=r" (end_high), "=r" (end_low) \
    : \
    : "%rax", "%rbx", "%rcx", "%rdx");
```

C

Model Specific Registers values for reverse-engineering the complex addressing function

C.1 Xeon CPUs

C.1.1 Monitoring session

To set up a monitoring session for the `LLC_LOOKUP` event on Xeon Sandy Bridge, Ivy Bridge and Haswell CPUs, the following steps must be taken:

1. freeze box counters,
2. reset counter and control registers,
3. enable counting,
4. select `LLC_LOOKUP` event,
5. select all MESIF states,
6. unfreeze box counters,
7. launch program to monitor,
8. freeze box counters,
9. read counter value in `PMON_CTR0`.

C.1.2 MSR addresses and values for Xeon Sandy Bridge CPUs

MSR addresses for Xeon Sandy Bridge CPUs can be found in [Int12]. The values are derived from the MSR layout and events.

Table C.1: MSR addresses – Xeon Sandy Bridge CPUs

	PMON_CTR0	PMON_BOX_FILTER	PMON_CTL0	PMON_BOX_CTL
CBo 0	0xd16	0xd14	0xd10	0xd04
CBo 1	0xd36	0xd34	0xd30	0xd24
CBo 2	0xd56	0xd54	0xd50	0xd44
CBo 3	0xd76	0xd74	0xd70	0xd64
CBo 4	0xd96	0xd94	0xd90	0xd84
CBo 5	0xdb6	0xdb4	0xdb0	0xda4
CBo 6	0xdd6	0xdd4	0xdd0	0xdc4
CBo 7	0xdf6	0xdf4	0xdf0	0xde4

Table C.2: MSR values for LLC_LOOKUP event monitoring – Xeon Sandy Bridge CPUs

value	**MSR**	**description**
0x10100	PMON_CTR0	freeze counters
0x10103	PMON_BOX_CTL	reset counter and control registers
0x400000	PMON_CTL0	enable counting
0x401134	PMON_CTL0	select LLC_LOOKUP event
0x7c0000	PMON_BOX_FILTER	select all MESIF states
0x10000	PMON_BOX_CTL	unfreeze counters

C.1.3 MSR addresses and values for Xeon Ivy Bridge CPUs

MSR addresses for Xeon Ivy Bridge CPUs can be found in [Int14c]. The values are derived from the MSR layout and events.

Table C.3: MSR addresses – Xeon Ivy Bridge CPUs

	PMON_CTR0	PMON_BOX_FILTER	PMON_CTL0	PMON_BOX_CTL
CBo 0	0xd16	0xd14	0xd10	0xd04
CBo 1	0xd36	0xd34	0xd30	0xd24
CBo 2	0xd56	0xd54	0xd50	0xd44
CBo 3	0xd76	0xd74	0xd70	0xd64
CBo 4	0xd96	0xd94	0xd90	0xd84
CBo 5	0xdb6	0xdb4	0xdb0	0xda4
CBo 6	0xdd6	0xdd4	0xdd0	0xdc4
CBo 7	0xdf6	0xdf4	0xdf0	0xde4
CBo 8	0xe16	0xe14	0xe10	0xe04
CBo 9	0xe36	0xe34	0xe30	0xe24
CBo 10	0xe56	0xe54	0xe50	0xe44
CBo 11	0xe76	0xe74	0xe70	0xe64
CBo 12	0xe96	0xe94	0xe90	0xe84
CBo 13	0xeb6	0xeb4	0xeb0	0xea4
CBo 14	0xed6	0xed4	0xed0	0xec4

Table C.4: MSR values for LLC_LOOKUP event monitoring – Xeon Ivy Bridge CPUs

value	**MSR**	**description**
0x30100	PMON_CTR0	freeze counters
0x30103	PMON_BOX_CTL	reset counter and control registers
0x400000	PMON_CTL0	enable counting
0x401134	PMON_CTL0	select LLC_LOOKUP event
0x7e0010	PMON_BOX_FILTER	select all MESIF states
0x30000	PMON_BOX_CTL	unfreeze counters

C.1.4 MSR addresses and values for Xeon Haswell CPUs

MSR addresses for Xeon Haswell CPUs can be found in [Int14d]. The values are derived from the MSR layout and events.

Table C.5: MSR addresses – Xeon Haswell CPUs

	PMON_CTR0	PMON_BOX_FILTER	PMON_CTL0	PMON_BOX_CTL
CBo 0	0xe08	0xe05	0xe01	0xe00
CBo 1	0xe18	0xe15	0xe11	0xe10
CBo 2	0xe28	0xe25	0xe21	0xe20
CBo 3	0xe38	0xe35	0xe31	0xe30
CBo 4	0xe48	0xe45	0xe41	0xe40
CBo 5	0xe58	0xe55	0xe51	0xe50
CBo 6	0xe68	0xe65	0xe61	0xe60
CBo 7	0xe78	0xe75	0xe71	0xe70
CBo 8	0xe88	0xe85	0xe81	0xe80
CBo 9	0xe98	0xe95	0xe91	0xe90
CBo 10	0xea8	0xea5	0xea1	0xea0
CBo 11	0xeb8	0xeb5	0xeb1	0xeb0
CBo 12	0xec8	0xec5	0xec1	0xec0
CBo 13	0xed8	0xed5	0xed1	0xed0
CBo 14	0xee8	0xee5	0xee1	0xee0
CBo 15	0xef8	0xef5	0xef1	0xef0
CBo 16	0xf08	0xf05	0xf01	0xf00
CBo 17	0xf18	0xf15	0xf11	0xf10

Table C.6: MSR values for LLC_LOOKUP event monitoring – Xeon Haswell CPUs

value	**MSR**	**description**
0x30100	PMON_CTR0	freeze counters
0x30103	PMON_BOX_CTL	reset counter and control registers
0x400000	PMON_CTL0	enable counting
0x401134	PMON_CTL0	select LLC_LOOKUP event
0x7e0020	PMON_BOX_FILTER	select all MESIF states
0x30000	PMON_BOX_CTL	unfreeze counters

C.2 Core CPUs

Uncore performance monitoring for Intel Core processors is described in [Int14b], Section 18.9.6.

C.2.1 Monitoring session

To set up a monitoring session for the event – called `UNC_CBO_CACHE_LOOKUP` for Core processors – the following steps must be taken:

1. disable counters,
2. reset counters,
3. select event to monitor,
4. enable counting,
5. launch program to monitor,
6. read counter value from `MSR_UNC_CBO_i_PER_CTR0` for each CBo i.

C.2.2 MSR addresses and values

MSR addresses for Core processors can be found in [Int14b], Section 35.8.1. The values are derived from the MSR layout Section 18.9.6 and from the events described in Section 19.4, Table 19-10. They are valid for Sandy Bridge, Ivy Bridge and Haswell microarchitectures.

Table C.7: MSR addresses – Core CPUs

MSR	**address**
`MSR_UNC_PERF_GLOBAL_CTRL`	`0x391`
`MSR_UNC_CBO_0_PERFEVTSEL0`	`0x700`
`MSR_UNC_CBO_0_PER_CTR0`	`0x706`
`MSR_UNC_CBO_1_PERFEVTSEL0`	`0x710`
`MSR_UNC_CBO_1_PER_CTR0`	`0x716`
`MSR_UNC_CBO_2_PERFEVTSEL0`	`0x720`
`MSR_UNC_CBO_2_PER_CTR0`	`0x726`
`MSR_UNC_CBO_3_PERFEVTSEL0`	`0x730`
`MSR_UNC_CBO_3_PER_CTR0`	`0x736`

Table C.8: MSR values for `UNC_CBO_CACHE_LOOKUP` event monitoring – Core CPUs

value	**MSR**	**description**
`0x2000000f`	`MSR_UNC_PERF_GLOBAL_CTRL`	enable counting
`0x0`	`MSR_UNC_PERF_GLOBAL_CTRL`	disable counting
`0x408f34`	`MSR_UNC_CBO_i_PERFEVTSEL0`	select `UNC_CBO_CACHE_LOOKUP` event
`0x0`	`MSR_UNC_CBO_i_PER_CTR0`	reset counters

List of Figures

List of Tables

Bibliography

[ABG10] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New Results on Instruction Cache Attacks. In *Proceedings of the 12th Workshop on Cryptographic Hardware and Embedded Systems (CHES'10)*, pages 110–124, 2010.

[Aci07] Onur Aciiçmez. Yet Another MicroArchitectural Attack: Exploiting I-cache. In *Proceedings of the 1st ACM Computer Security Architecture Workshop (CSAW'07)*, pages 11–18, 2007.

[AGS07] Onur Aciiçmez, Shay Gueron, and Jean-pierre Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, 2007.

[AHFG10] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2nd ACM Cloud Computing Security Workshop (CCSW'10)*, pages 103–108, 2010.

[AK06] Onur Aciiçmez and Çetin Kaya Koç. Trace-Driven Cache Attacks on AES (Short Paper). In *Proceedings of the 8th international conference on Information and Communications Security*, pages 112–121, 2006.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security (ASIACCS'07)*, pages 312–320, 2007.

[Amaa] Amazon Web Services. Amazon EC2 instance types. `https://aws.amazon.com/ec2/instance-types/`. Retrieved on August 30, 2015.

[Amab] Amazon Web Services. Linux GPU Instances. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using_cluster_computing.html`. Retrieved on August 30, 2015.

[AS07] Onur Aciiçmez and Jean-Pierre Seifert. Cheap Hardware Parallelism Implies Cheap Security. *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)*, pages 80–91, September 2007.

[ASK07a] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache Based Remote Timing Attack on the AES. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA 2007)*, pages 271–286, 2007.

[ASK07b] Onur Aciiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA 2007)*, pages 225–242, 2007.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago, 2005.

[BGNS06] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *Cryptology ePrint Archive, Report 2006/052*, 2006.

[BJB15] Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and Efficient Elimination of Cache and Timing Side Channels. *arXiv:1506.00189*, 2015.

[BK07] Johannes Blömer and Volker Krummel. Analysis of countermeasures against access driven cache attacks on AES. In *Proceedings of the 14th international conference on Selected areas in cryptography (SAC'07)*, pages 96–109, 2007.

[BKS13] Sebastian Breß, Stefan Kiltz, and Martin Schäler. Forensics on GPU Coprocessing in Databases – Research Challenges, First Experiments, and Countermeasures. In *Proceedings of the 1st Workshop on Databases in Biometrics, Forensics and Security Applications*, pages 115–129, 2013.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Proceedings of the 8th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2006)*, pages 1–16, 2006.

[BMPP12] Adam Bates, Benjamin Mood, Joe Pletcher, and Hannah Pruse. Detecting Co-Residency with Active Traffic Analysis Techniques. In *Proceedings of the 4th ACM Cloud Computing Security Workshop (CCSW'12)*, 2012.

[BRPG15] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R Gross. CAIN: Silently Breaking ASLR in the Cloud. In *Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT'15)*, 2015.

[BSS10] A Bernemann, R Schreyer, and K Spanderen. Pricing structured equity products on GPUs. In *Proceedings of the 2010 IEEE Workshop on High Performance Computational Finance*, pages 1–7, 2010.

[BvdPSY14] Naomi Benger, Joop van de Pool, Nigel P. Smart, and Yuval Yarom. "Ooh Aah... Just a Little Bit" : A small amount of side channel can go a long way. In *Proceedings of the 16th Workshop on Cryptographic Hardware and Embedded Systems (CHES'14)*, pages 75–92, 2014.

[BZB$^+$05] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, volume 2, pages 586–591 Vol. 1. Ieee, 2005.

[CCD12] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–20, 2012.

[CH11] Yves Crama and Peter L Hammer. *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press, 2011.

[CHB$^+$15] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[CNZ$^+$11] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 189–202, 2011.

[CVDD09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (S&P'09)*, volume 45-60, 2009.

[CVE12a] CVE-2012-0946. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0946`, 2012.

[CVE12b] CVE-2012-4225. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4225`, 2012.

[CVE13a] CVE-2013-0109. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0109`, 2013.

[CVE13b] CVE-2013-0110. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0110`, 2013.

[CVE13c] CVE-2013-0131. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0131`, 2013.

[CVE15] CVE-2015-3456. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456`, 2015.

[CYC$^+$01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *ACM SIGOPS Operating Systems Review*, 35(5):73, 2001.

[DFK$^+$13] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proceedings of the 22nd USENIX Security Symposium*, pages 431–446, 2013.

[DJL$^+$11] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-ghazaleh, and Dmitry Ponomarev. A Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4), 2011.

[DLV13] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. CUDA Leaks: Information Leakage in GPU Architectures. *arXiv:1305.7383v1*, 2013.

[DMS$^+$13] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559–570, 2013.

[DPS13] Janis Danisevskis, Marta Piekarska, and Jean-Pierre Seifert. Dark Side of the Shader: Mobile GPU-aided Malware Delivery. In *Proceedings of the 16th Annual International Conference on Information Security and Cryptology (ICISC'13)*, 2013.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 226–231, 1996.

[Env] Envytools. `https://github.com/envytools/envytools`.

[FL15] Adi Fuchs and Ruby 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)*, 2015.

[For14] Forbes. Intel And AMD: The Juggernaut Vs. The Squid. `http://www.forbes.com/sites/rogerkay/2014/11/25/intel-and-amd-the-juggernaut-vs-the-squid/`, November 2014. Retrieved on September 6, 2015.

[GBK11] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P'11)*, pages 490–505, May 2011.

[GBM12] Johannes Gilger, Johannes Barnickel, and Ulrike Meyer. GPU-Acceleration of Block Ciphers in the OpenSSL Cryptographic Library. In *Proceedings of the 15th international conference on Information Security (ISC'12)*, pages 338–353, 2012.

[GBM15] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS'15)*, 2015.

[gE] gKrypt Engine. `http://gkrypt.com/`.

[GMM15] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *arXiv:1507.06955v1*, 2015.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[Hu92] Wei-Ming Hu. Lattice Scheduling and Covert Channels. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 52–61, 1992.

[HW09] Owen Harrison and John Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In *Proceedings of the Second International Conference on Cryptology in Africa (AFRICACRYPT 2009)*, 2009.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P'13)*, pages 191–205. Ieee, May 2013.

[IES15a] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Lucky 13 Strikes Back. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'15)*, pages 85–96, 2015.

[IES15b] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing—and its Application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[IES15c] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *Proceedings of the 18th EUROMICRO Conference on Digital System Design*, 2015.

[IIES14a] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain Cross-VM Attacks on Xen and VMware are possible! *Cryptology ePrint Archive, Report 2014/248*, 2014.

[IIES14b] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, Cross-VM attack on AES. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*, 2014.

[IIES15] Gorka Irazoqui, Mehmet Sinan IncI, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. *Proceedings on Privacy Enhancing Technologies*, 1(1):25–40, 2015.

[Int08] Intel. Advanced Encryption Standard (AES) Instructions Set: White Paper, 2008.

[Int10] Intel. How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper, 2010.

[Int12] Intel. Intel ® Xeon ® Processor E5-2600 Product Family Uncore Performance Monitoring Guide. 327043-001:1–136, 2012.

[Int14a] Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2014.

[Int14b] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 3(253665), 2014.

[Int14c] Intel. Intel® Xeon® Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual. 329468-002:1—–200, 2014.

[Int14d] Intel. Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Reference Manual. 331051-001:1–232, 2014.

[JGSW] Sanjeev Jahagirdar, Varghese George, Inder Sodhi, and Ryan Wells. Power Management of the Third Generation Intel Core Micro Architecture formerly codenamed Ivy Bridge. Hot Chips 2012, `http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf`. Retrieved on July 16, 2015.

[JTSE10] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News*, 38(3):60, 2010.

[KÖ8] Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA 2008)*, pages 187–202, 2008.

[KASZ08] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM Computer Security Architectures Workshop (CSAW'08)*, New York, New York, USA, 2008. ACM Press.

[KASZ09] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA'09)*, pages 393–404, 2009.

[KASZ13] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Architecting against Software Cache-Based Side-Channel Attacks. *IEEE Transactions on Computers*, 62(7):1276–1288, 2013.

[Ker12] Michael Kerrisk. Xdc2012: Graphics stack security. `https://lwn.net/Articles/517375/`, 2012.

[KKL$^+$07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm : the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[KMMB12] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, 2012.

[KNQ14] Dae-hyun Kim, Prashant J Nair, and Moinuddin K Qureshi. Architectural Support for Mitigating Row Hammering in DRAM Memories. *IEEE Computer Architecture Letters*, 14(1):9–12, 2014.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference (Crypto'96)*, pages 104–113, 1996.

[Kol] Con Kolivas. cgminer. `https://github.com/ckolivas/cgminer`.

[KP05] Craig Kolb and Matt Pharr. *GPU Gems 2*, chapter Options Pricing on the GPU. 2005.

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In *Proceedings of the 11th Workshop on Cryptographic Hardware and Embedded Systems (CHES'09)*, 2009.

[Lau05] Cédric Lauradoux. Collision attacks on processors with cache and countermeasures. In *Proceedings of Western European Workshop on Research in Cryptology (WEWoRC 2005)*, pages 76–85, 2005.

[LGR13] Peng Li, Debin Gao, and Michael K. Reiter. Mitigating Access-Driven Timing Channels in Clouds Using StopWatch. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, pages 1–12, 2013.

[LKKK14] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[LKV$^+$13] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec'13)*, 2013.

[LL13] Fangfei Liu and Ruby B. Lee. Security testing of a secure cache design. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*, pages 1–8, 2013.

[LL14] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, pages 203–215, 2014.

[LLND10] Fernand Lone Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE'10)*, pages 7–14, 2010.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[LP10] Flavio Lombardi and Roberto Di Pietro. CUDACS: Securing the Cloud with CUDA-Enabled Secure Virtualization. In *Proceedings of the 12th International Conference on Information and Communications Security (ICICS'10)*, pages 92–106, 2010.

[LWL15] Fangfei Liu, Hao Wu, and Ruby B. Lee. Can randomized mapping secure instruction caches from side-channel attacks? In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP'15)*, number August, 2015.

[LYG$^+$15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-Level Cache Side-Channel Attacks are Practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[MDS12] Robert Martin, John Demme, and Simha Sethumadhavan. TimeWarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA'12)*, pages 118–129, 2012.

[MKS10] Tomosuke Murakami, Ryuta Kasahara, and Takamichi Saito. An implementation and its evaluation of password cracking tool parallelized on GPGPU. In *Proceedings of the 10th International Symposium on Communications and Information Technologies*, pages 534–538. Ieee, October 2010.

[MKS12] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. Are AES x86 Cache Timing Attacks Still Feasible? In *Proceedings of the 4th ACM Cloud Computing Security Workshop (CCSW'12)*, pages 19–24, 2012.

[MLSN+15] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel's Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'15)*, November 2015.

[MNHF14] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Confidentiality Issues on a GPU in a Virtualized Environment. In *Proceedings of the 18th International Conference on Financial Cryptography and Data Security (FC'14)*, March 2014.

[MNHF15] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, July 2015.

[MON+13] Clémentine Maurice, Stéphane Onno, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Improving 802.11 Fingerprinting of Similar Devices by Cooperative Fingerprinting. In *Proceedings of the 2013 International Conference on Security and Cryptography (SECRYPT'13)*, August 2013.

[Moo75] Gordon E. Moore. Progress in digital integrated electronics. *IEEE International Electron Devices Meeting*, pages 11–13, 1975.

[MZK11] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. In *Proceedings of the sixth ACM workshop on Scalable Trusted computing*, 2011.

[Nou] Nouveau. `http://nouveau.freedesktop.org`.

[NS06] Michael Neve and Jean-Pierre Seifert. Advances on Access-Driven Cache Attacks on AES. In *Proceedings of the 13th international conference on Selected areas in cryptography (SAC'06)*, pages 147–162, 2006.

[NSW06] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at Bernstein's AES side-channel analysis. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security (ASIACCS'06)*, 2006.

[NVI] NVIDIA. NVIDIA GRID GPU specs and features. `http://www.nvidia.com/object/cloud-gaming-gpu-boards.html`. Retrieved on August 30, 2015.

[NVI10] NVIDIA. TESLA M2050 / M2070 GPU computing module, 2010.

[NVI12] NVIDIA. CUDA C Programming Guide, 2012.

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, 2015.

[Ope] OpenSSL. Changelog. `http://www.openssl.org/news/changelog.html`.

[Ope13] OpenStack. OpenStack Security Guide. page 238, 2013.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA 2006)*, pages 1–25, 2006.

[OW11] Rodney Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *Proceedings of the 30th IEEE International Performance Computing and Communications Conference (IPCCC'11)*, pages 1–8, November 2011.

[Pag02] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical report, Department of Computer Science, University of Bristol, 2002.

[Pag05] Dan Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint Archive, Report 2005/280*, page 14, 2005.

[Pat] Pathscale. pscnv - PathScale NVIDIA graphics driver. `https://github.com/pathscale/pscnv`.

[Per05] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, pages 1–13, 2005.

[PLS$^+$14] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'14)*, pages 305–316, 2014.

[QJP+07] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381, June 2007.

[RNSE09] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource Management for Isolation Enhanced Cloud Services. In *Proceedings of the 1st ACM Cloud Computing Security Workshop (CCSW'09)*, pages 77–84, 2009.

[RSD06] Chester Rebeiro, David Selvakumar, and A.S.L. Devi. Bitslice Implementation of AES. In *Proceedings of the 5th International Conference on Cryptology and Network Security (CANS'06)*, pages 203–212, 2006.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM conference on Computer and Communications Security (CCS'09)*, pages 199–212, 2009.

[Sea15a] Mark Seaborn. Exploiting the DRAM rowhammer bug to gain kernel privileges. `http://googleprojectzero.blogspot.fr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`, March 2015. Retrieved on June 2, 2015.

[Sea15b] Mark Seaborn. L3 cache mapping on Sandy Bridge CPUs. `http://lackingrhoticity.blogspot.fr/2015/04/l3-cache-mapping-on-sandy-bridge-cpus.html`, April 2015. Retrieved on June 2, 2015.

[SIYA11] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication as a Threat to the Guest OS. In *Proceedings of the 4th European Workshop on System Security*, 2011.

[SKLR11] Jakub Szefer, Eric Keller, Ruby B Lee, and Jennifer Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud Categories and Subject Descriptors. In *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS'11)*, pages 401–412, 2011.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *ACM SIGOPS Operating Systems Review*, 41(6):335–350, 2007.

[SMA09] Marco Slaviero, Haroon Meer, and Nick Arvanitis. Clobbering the Cloud! *Black Hat 2009 Briefings*, 2009. Retrieved on August 26, 2015.

[Smo09] Christopher Smowton. Secure 3D graphics for virtual machines. In *Proceedings of the 2nd European Workshop on System Security (EUROSEC'09)*, pages 36–43, 2009.

[SP13] Raphael Spreitzer and Thomas Plos. Cache-access pattern attack on disaligned AES t-tables. In *Proceedings of the 4th international conference on Constructive Side-Channel Analysis and Secure Design (COSADE'13)*, pages 200–214, 2013.

[Spr11] Martijn Sprengers. *GPU-based Password Cracking*. Master of science thesis, Radboud University Nijmegen, 2011.

[SSCZ11] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSNW'11)*, pages 194–199, 2011.

[SXZ13] Brendan Saltaformaggio, Dongyan Xu, and Xiangyu Zhang. BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels. In *Proceedings of the 6th European Workshop on Systems Security (EuroSec'13)*, 2013.

[TB10] Xiang Tian and Khaled Benkrid. High-performance quasi-monte carlo financial simulation: FPGA vs. GPP vs. GPU. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(4), 2010.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 23(1):37–71, July 2010.

[TSS03] Yukiyasu Tsunoo, Teruo Saito, and Tomoyasu Suzaki. Cryptanalysis of DES implemented on computers with cache. In *Proceedings of the 5th Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)*, pages 62–76, 2003.

[UGV08] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *Proceedings of the 5th International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2008)*, pages 59–67, 2008.

[VAP$^+$08] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, pages 116–134, 2008.

[VDS11] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW'11)*, pages 41–46, 2011.

[VMw09a] VMware. Performance Evaluation of AMD RVI Hardware Assist. Technical report, 2009.

[VMw09b] VMware. Performance Evaluation of Intel EPT Hardware Assist. Technical report, 2009.

[VPA+09] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID'09)*, pages 265–283, 2009.

[VPAI14] Giorgos Vasiliadis, Michalis Polychronakis, Elias Athanasopoulos, and Sotiris Ioannidis. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*, 2014.

[VPI10] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. GPU-assisted malware. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software*, pages 1–6, 2010.

[VRS14] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based Defenses against Cross-VM Side-channels. In *Proceedings of the 23th USENIX Security Symposium*, 2014.

[VZRS15] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *Proceedings of the 24th USENIX Security Symposium*, pages 913–928, 2015.

[WHF+12] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, and Thorsten Holz. Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, pages 189–198, 2012.

[WHS12] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A Cache Timing Attack on AES in Virtualization Environments. In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security (FC'12)*, number 01, pages 314–328, 2012.

[WL06] Zhenghong Wang and Ruby B Lee. Covert and Side Channels due to Processor Architecture. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 473–482, 2006.

[WL07] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2):494, June 2007.

[WL08] Zhenghong Wang and Ruby B. Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*, pages 83–93, 2008.

[Won13] Henry Wong. Intel Ivy Bridge Cache Replacement Policy. `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`, 2013. Retrieved on July 22, 2015.

[WR11] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software attacks against Intel VT-d technology. *invisiblethingslab.com*, 2011.

[WRC08] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC'08)*, pages 15–28, 2008.

[WXW12] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[WXW14] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. *IEEE/ACM Transactions on Networking*, 2014.

[XBJ$^+$11] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW'11)*, pages 29–40, 2011.

[XLCZ12] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 42th International Conference on Dependable Systems and Networks (DSN'12)*, pages 1–12, 2012.

[XWW15] Zhang Xu, Haining Wang, and Zhenyu Wu. A Measurement Study on Co-residence Threat inside the Cloud. In *Proceedings of the 24th USENIX Security Symposium*, pages 929–944, 2015.

[XXHW13] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, pages 1–12. Ieee, June 2013.

[Yam07] Takeshi Yamanouchi. *GPU Gems 3*, chapter AES Encryption and Decryption on the GPU. 2007.

[YB14] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-channel Attack. *Cryptology ePrint Archive, Report 2014/140*, 2014.

[YF14] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23th USENIX Security Symposium*, 2014.

[ZJOR11] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Home-Alone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P'11)*, pages 313–328, May 2011.

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS'12)*, 2012.

[ZJRR14] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, pages 990–1003, New York, New York, USA, 2014. ACM Press.

[ZR13] Yinqian Zhang and Michael K. Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CCS'13)*, pages 827–838, 2013.