Benjamin Semal

Microarchitectural Covert Channels in Multitenant Computing Environments

Thesis submitted for the degree of Doctor of Philosophy to the

Royal Holloway University of London

2021

Declaration of Authorship

This doctoral study was conducted under the supervision of Professor Konstantinos Markantonakis.

The work presented in this thesis is the result of original research carried out by myself whilst enrolled in the Information Security Group as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Signature

Date

Abstract

The elastic property of cloud services relies on a dynamic mapping between distinct virtual terminals and shared physical nodes, laying bare correlations between the activity of concurrent tenants and the availability of microarchitectural resource. In this context, this thesis studies the threat posed by microarchitectural covert channels to data confidentiality in multi-tenant computing environments. An empirical study is conducted on the practicality of theses attacks against public Infrastructure-as-a-Service instances, revealing that covert channels achieve a medium severity score with the Common Vulnerability Scoring System. A new evaluation framework is then developed so as to devise metrics for fair comparison and identify conditions for eluding logical isolation on contemporary computing environments. As a result, two new microarchitectural covert channels based on Intel's integrated memory controllers are presented, which enable circumventing existing defense strategies. The first attack allows a privileged adversary to leak information between two processes within a single native environment. The second attack is an extension to cross-VM scenarios for unprivileged adversaries. An exhaustive study on state-of-the-art countermeasures is then realised, revealing a lack of perspective in their design approach. The analysis leads to a new covert channel based on Intel and AMD memory bus implementations. The resulting attack is tested across two AWS EC2 instances, demonstrating that an malicious individual can easily make his way around all existing countermeasures proposed in academia.

Acknowledgements

I would like to acknowledge and thank the many people who have helped me bringing this thesis to completion.

My initial and foremost thanks goes to Maria for her unwavering confidence in my ability to undertake the PhD. My gratitude also goes to my family for their support and many advice throughout the past three years (and before), and to my newly found family for their trust and caring.

My very special thanks to Raja and Carlton for the invaluable help, guidance, and moral support all along the PhD. I would also like to express my gratitude to my supervisors, Konstantinos, Peter, and Keith, as well as other members of the laboratory, Darren, and Nicola. Many thanks to Jan for his valuable reviews and sense of humour. Finally, I would like to extend my appreciation to Royal Holloway and its members, and to everyone who took part in this work, directly or indirectly.

The PhD is indeed a marathon, and it would not have been possible without your help. Thank you all.

List of Publications

Publications directly related to this thesis.

Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. Leaky controller: cross-VM memory controller covert channel on multi-core systems. *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 3-16. Springer, 2020.

Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. A study on microarchitectural covert channel vulnerabilities in Infrastructure-as-a-Service. *International Conference on Applied Cryptography and Network Security*, pages 360-377. Springer, 2020.

Benjamin Semal, Konstantinos Markantonakis, Keith Mayes, and Jan Kalbantner. One covert channel to rule them all: a practical approach to data exfiltration in the cloud. *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 328-336. IEEE, 2020.

Other publications.

Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram. A certificateless group authenticated key agreement protocol for secure communication in untrusted UAV networks. *2018 IEEE/AIAA 37th Digital Avionics Systems Conference*, pages 1-8. IEEE, 2018.

Jan Kalbantner, Konstantinos Markantonakis, Darren Hurley-Smith, Raja Naeem Akram, and Benjamin Semal. P2PEdge: a decentralised, scalable P2P architecture for energy trading in real-time. *Energies*, 14(3):606, 2021.

Jan Kalbantner, Konstantinos Markantonakis, Darren Hurley-Smith, Carlton Shepherd, and Benjamin Semal. A DLT-based smart contract architecture for atomic and scalable trading. *arXiv preprint arXiv:2105.02937*, 2021.

Carlton Shepherd, Jan Kalbantner, Benjamin Semal, and Konstantinos Markantonakis. A side-channel analysis of sensor multiplexing for covert channels and application fingerprinting on mobile devices. *arXiv preprint arXiv:2110.06363*, 2021.

Carlton Shepherd, Jan Kalbantner, Benjamin Semal, and Konstantinos Markantonakis. Exploiting sensor multiplexing for covert channels and application fingerprinting on mobile devices. *arXiv preprint arXiv:2110.06363*, 2021.

Carlton Shepherd, Benjamin Semal, and Konstantinos Markantonakis. Investigating Black-Box Function Recognition Using Hardware Performance Counters. *arXiv preprint arXiv:2204.11639*, 2022.

Contents

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
AMI	Amazon Machine Image
API	Application Programming Interface
ARM	Advanced RISC Machine
AVX	Advanced Vector Extensions
AVX2	Advanced Vector Extensions 2
AWS	Amazon Web Service
BIOS	Basic Input Output System
BTB	Branch Target Buffer
CAT	Cache Allocation Technology
CCSS	Covert Channel Scoring System
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DES	Data Encryption Standard
DDR3	Double Data Rate 3
DDR4	Double Data Rate 4
DIMM	Dual-Inline Memory Module
DRAM	Dynamic Random Access Memory
EC2	Elastic Compute Cloud
ECC	Elliptic Curve Cryptography
EPSS	Exploit Prediction Scoring System
FPU	Floating Point Unit
FSB	Front-Side Bus
Gb	Gigabit
GB	Gigabyte
GCE	Google Cloud Engine

GDPR	General Data Protection Regulation
HC	Hardware Configuration
IaaS	Infrastructure-as-a-Service
IBM	International Business Machines
IN	Initialisation
ISA	Instruction Set Architecture
Kb	Kilobit
KB	Kilobyte
L1-D	Level 1 Data Cache
L1-I	Level 1 Instruction Cache
L2	Level 2 Cache
LLC	Last-Level Cache
LSU	Load-Store Unit
Mb	Megabit
MB	Megabyte
MBA	Memory Bandwidth Allocation
MMU	Memory Management Unit
MOB	Memory Order Buffer
NUMA	Non-Uniform Memory Access
OS	Operating System
PaaS	Platform-as-a-Service
PC	Program Counter
PCIe	Peripheral Component Interconnect Express
PR	Privilege Required
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RL	Remediation Level
ROP	Return Oriented Programming
RSA	Rivest Shamir Adleman
RSB	Return-Stack Buffer
SaaS	Software-as-a-Service
SATA	Serial AT Attachment
SGX	Software Guard eXtension
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multi-Threading
SRAM	Static Random Access Memory
SSE	Streaming SIMD Extensions
SSH	Secure Shell
SSL	Secure Sockets Layer
SSVC	Stakeholder-Specific Vulnerability Categorization
TDES	Triple-Data Encryption Standard

TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
TSC	Time-Stamp Counter
TSX	Transactional Synchronization Extension
USD	U.S. Dollar
VM	Virtual Machine
VPC	Virtual Private Cloud
WC	Write-Combining
μOPs	Micro-Operations

Chapter 1

Introduction

Contents

This chapter exposes the challenges faced in cloud computing ecosystems and presents the thesis' rationale along with its contributions. The scope of the thesis is then delimited. The reader will also find a discussion on the structure of this manuscript.

1.1 Setting the Scene

Today's notion of *cloud computing* dates back from the 1950s, when large-scale computing mainframes were introduced. The mainframe would provide a single, powerful computing environment to multiple users. In the 1960s, the Massachusetts Institute of Technology sought for new computers capable of more than one simultaneous user. The stimuli lead International Business Machines (IBM) to design the VM operating system [35], capable of dividing storage and system resources. The technology enabled a single mainframe to have multiple *virtual machines* (VMs) running concurrently, thus allowing distinct computing environments to co-exist on a single physical node. VMs provided better security since each user was running in its own operating system (OS), and improved reliability as no one user could take down the entire system. Resource usage could also be optimised and return on investment improved, thus making the technology accessible to a wider public. This early form of cloud computing was eventually galvanised by the advent of the Internet in the 1990s. With the costs of hardware coming down as well as a rocketing demand, corporations needed to combine multiple physical nodes together. Single software components, known as *orchestrators*, allowed managing a pool of hardware servers as if they were a single physical node, and enabled coping with the new demand.

Cloud computing then morphed into multiple shapes, with a cloud service for every need. Anyone who doesn't possess its own server can now rent an *instance* to a cloud provider, and access a seemingly unlimited pool of resources instantaneously. In 2018, it is estimated that 26% of the European Union's enterprises were dependent on cloud services, with a 21% increase in large corporations since 2014 [39]. Furthermore, the study reports that out of these companies relying on cloud computing, 55% use it for financial and accounting software applications, customer relationship management, or business applications that require advanced computational power. The data stored and processed on these cloud platforms can be of sensitive nature, which raises confidentiality issues. IBM Security evaluated the average cost of a data breach to 3.58 million USD, with 80% of the compromised data

being customer personally identifiable information [130].

Since the 1990s, researchers investigated the security aspects of multitenancy [61, 62, 163]. When multiple software components (threads, processes, or VMs) execute simultaneously, they compete with each other for processor resources. Such conflicts can delay the execution of certain instructions, resulting in timing variations to occur. These timing variations can in turn lead to accidental disclosure of secrets, such as cryptographic keys. Despite efforts to raise awareness over the risks associated with hardware optimisation [173, 16, 145, 50], manufacturers carried on regardless of security. A security gap grew progressively which eventually lead to unfortunate breakthroughs. The field of *microarchitectural attacks* entered its apogee with the release of the Spectre [81] and Meltdown [89] attacks in January 2018. Computer security, which heavily relies on memory isolation, was met with vulnerabilities that could only be addressed by modifying the microarchitecture, either in the form of microcode updates, or new hardware designs. Hardware manufacturers could no longer fall back on application developers to take the responsibility. These new security bugs opened the way to an avalanche of disclosures on newly found vulnerabilities, challenging the foundations on which the security of our computers is based.

1.2 Motivation and Challenges

Microarchitectural timing-based attacks exploit the sharing of a processor among multiple tenants, in order to compromise sensitive information. Thus they have the potential to severely compromise the viability of the multitenant computing model. In 2013, the Cloud Security Alliance rated cross-VM leakage channels as the number one threat to cloud computing [11]. Two colluding entities can intentionally create timing variations in order to encode and decode binary information. Based on this principle, microarchitectural *covert channel attacks* allow tunneling information out of a compromised system when the security policy does not allow doing so. In other words, an attacker can extract information covertly out of a victim's cloud instance. Significant work has been accomplished in the field of microarchitectural at-

tacks, however it almost exclusively focuses on side channels. Also, covert channels are often associated with the hiding of secret information in network protocols. Microarchitectural covert channels have a powerful potential which draws from both disciplines.

Microarchitectural covert channels offer a quieter means to exfiltrate data than network protocols, although they rely on a stronger adversary model. They assume an attacker with advanced capabilities that seeks to maintain long-term access to the victim's instance. This type of adversary has other incentives than simple financial gain or notoriety. Indeed, the IBM Security study reports that 13% of data breaches are carried out by nation state actors [130], which have a more strategic approach to cyber-warfare. Yet, it is not trivial to assess the severity and practicality of these attacks. Although the latest version of the Common Vulnerability Scoring System (CVSS) provides more flexibility, it is still not adequate to assess the severity microarchitectural covert channels. Furthermore, these attacks are not well understood by the community. Security experts tend to associate microarchitectural attacks to side channels, which rely on a substantially different adversary model.

Microarchitectural attacks exploit hardware vulnerabilities that are specific to a processor implementation, also know as a microarchitecture, from the software domain. An attacker that intends to modulate a microarchitectural component must do so at a very low level. The inner mechanics of the Central Processing Unit (CPU) must be well understood. There is a significant amount of operations which are transparent to the developer, and over which the attacker must have fine-grained control, such as memory addressing, pipeline execution, caching, out-of-order execution, dynamic RAM (DRAM) refreshing, etc. In 2020, it is estimated that x86 processors represented 90% of the global revenue generated by the market of cloud servers [64]. Therefore, the vulnerabilities of interest are most often specific to Intel and AMD processors. Beyond the adversary model, the challenge to deploying these attacks is to gain detailed knowledge of the microarchitecture despite the lack of public information provided by hardware manufacturers. For instance, Intel does not reveal its DRAM addressing functions, forcing attackers to undertake reverse engineering studies.

In a context where cloud consumers are facing major cyber-threats, this thesis investigates security guarantees offered by virtualisation on data confidentiality, by exploring new means of compromising logical isolation based on x86 hardware vulnerabilities. Ultimately, it is hoped that this work will further our understanding of complex processor architectures, expose certain security risks oblivious to most of cloud consumers, and encourage considerations for security when designing future processor generations.

1.3 Contributions

The contribution of this thesis is threefold. First, a study is conducted on the operational constraints related to the deployment of microarchitectural covert channels in Infrastructure-as-a-Service (IaaS), based on an analysis of state-of-the-art attacks. The adversary model is studied under the CVSS v3.1 [113], with a resulting severity score of 5.0. In comparison, the SSLv3 POODLE [110] and OpenSSL Heartbleed [109] vulnerabilities achieved respective scores of 3.4 and 7.5. This research thread is the first illustration of the vulnerability of multitenant computing environments to covert channel attacks. Furthermore, a bespoke framework is devised which includes three additional evaluation criteria. The resulting Covert Channel Scoring System (CCSS) is capable of outlining disparities among attacks which the CVSS does not. Eventually, this study reveals the existence of a gap between the direction taken by researchers, and the actual challenges faced by the industry. For instance, the analysis shows that cache-based covert channels, although increasingly popular, are the least viable attacks. The CCSS may serve as a metric for the performance of future publications, by outlining whether the covert channel attack can be deployed in a commercial environment. Eventually, this research thread successfully refutes the erroneous intuition over the lack of practicality of microarchitectural covert channels, and puts forward a new method for evaluating the practicality of these attacks.

Second, a vulnerability on Intel's integrated memory controllers is exposed. For the first time, an instantiation of a covert channel attack is proposed, both in native and virtualised environments. The cross-VM covert

channel is tested across three different Intel microarchitectures, namely Ivy Bridge, Broadwell, and Skylake. The cross-core covert channel achieves a capacity of 729 bps in a native scenario, and up to 95 bps in the cross-VM scenario. This work outperforms other attacks which require the enabling of page sharing, simultaneous multi-threading (SMT), or which can be thwarted by the myriad of existing countermeasures [33, 37, 53, 62, 78, 90, 99, 152, 155, 168]. This research thread successfully demonstrates the presence of remaining vulnerabilities on x86 processors, and that there are alternatives to cache-based timing channels. Potential countermeasures are finally suggested.

Third, a new instance of the memory bus-based covert channel is devised, circumventing every relevant leakage channel countermeasure proposed in academia so far. Indeed, significant efforts have been made in order to address timing channel vulnerabilities, however new attacks tend to disregard the relevant countermeasures, arguing that these have not yet been deployed by OS and cloud providers. This research thread proposes a retrospective analysis on state-of-the-art attack and defence techniques, and shows that all existing covert channels could effectively be closed. The x86 memory bus vulnerability is re-visited in order to discard the usage of artifacts which are theoretically made unavailable by recently proposed countermeasures. The new attack is then deployed on the Amazon Web Service (AWS) Elastic Compute Cloud (EC2) commercial IaaS platform. The new instance of the memory bus-based covert channel effectively demonstrates that x86 microarchitectures (Intel and AMD) still present salient vulnerabilities, and that state-of-the-art defence strategies—even theoretical ones—remain unsuccessful at hindering data leakage in multi-tenant environments.

1.4 Scope

Microarchitectural covert channels exploit vulnerabilities in the implementation of a processor's architecture. In contrast, network covert channels abuse network protocols [161, 145]. This thesis examines the side-effects of multitenancy on the confidentiality of cloud-based data and services, therefore

the latter case is not consistent with the thesis rationale. Also, not all covert channels have malicious intents. For example, some will be leveraged to add authentication features over existing industrial networks. In this thesis, the term *covert channel* refers exclusively to attacks.

The term *leakage channel* encompasses both side and covert channels, which differ in the attack scenario. Independently from the adversary model, leakage channels share the underlying mechanisms. The distinction lies in whether timing variations are generated accidentally or intentionally. In addition, some related works classify microarchitectural attacks between storage-based and timing-based channels. A storage-based channel utilises variations in data values as the sole mechanism to encode and interpret information. In contrast, a timing-based channel exploits the time to access this data. In a virtualised ecosystem, only the latter case is relevant.

Finally, a covert channel does not necessarily allow communication across VMs. However, this thesis investigates covert channel attacks in the context of multitenant environments, i.e. virtualised. Therefore, only *cross-VM* covert channels are considered. This study could be applied to environments similar to IaaS, e.g. private clouds. It is worth mentioning that the evaluation of the adversary model presented in Chapter 3 is specific to IaaS, and does not account for variations in similar environments.

1.5 Thesis Outline

Chapter 2 concentrates the background related to subsequent chapters, from cloud computing models, through hardware architecture, to logical isolation principles. A literature review is then provided on the field of microarchitectural attacks along with a presentation of covert channel attacks.

Chapter 3 studies the adversary model of microarchitectural covert channel in a cloud environment, and demonstrates the severity of such attacks. A bespoke evaluation framework is then proposed which outperforms the CVSS, notably when comparing one covert channel implementation to another.

Chapter 4 presents a new covert channel attack which leverages timing vulnerabilities in Intel memory controllers. The attack allows leaking infor-

mation across processes with distinct address spaces, and across VMs (at reduced performance), thus offering an alternative to similar attacks based on the sole exploitation of cache memories. The covert channel is tested in a laboratory environment.

Chapter 5 presents a new technique to elude timing channel countermeasures. It is based on a memory bus covert channel which leverages timing variations incurred by atomic accesses to cache line-crossing memory regions. The attack is tested on both laboratory and commercial environments, namely the Amazon Web Service Elastic Compute Cloud service.

Chapter 6 concludes on the contributions of this thesis and identifies potential research threads related to this thesis.

Chapter 2

Background

Contents

This chapter introduces cloud computing models with a discussion on the current state of the market. Underlying concepts to computer architecture are then presented. Finally, a literature review on microarchitectural transient and timing attacks is provided, with an emphasis on covert channels.

2.1 Cloud Ecosystem

Cloud computing is an ubiquitous and yet equivocal term. One can summarise it as the delivery of different services through the Internet. This section defines cloud services under different models, and provides insights into the state of the cloud market.

2.1.1 Overview of Cloud Models

The NIST defines three types of cloud services [102], namely Infrastructure-as-a-Service, Platform-as-a-Service, and Software-as-a-Service. Although not exhaustive, this definition of cloud services is widely adopted in the community, and will therefore be used in the remainder of this thesis. Table 2.1 provides a comparison of resource ownership among the three cloud categories.

Software-as-a-Service (SaaS) offers internet-based software applications to end-users. A characteristic of SaaS is that all resources are managed by the remote third-party. The end-user is not required to install any software on its machine, and security is taken care of by the service provider. Services are typically accessed via a web browser. Examples of SaaS include Microsoft Office 365, Dropbox, Google Docs, etc.

Platform-as-a-Service (PaaS) consists of an internet-accessible environment for developing, maintaining, and executing software applications. End-users (mostly developers) are provided with a dedicated runtime environment, and are responsible for managing the software application and its associated data. Examples of PaaS include Google App Engine, Amazon Web Service Elastic Beanstalk, Red Hat OpenShift, etc.

Infrastructure-as-a-Service delivers internet-accessible storage, processing, and network resources. The end-user controls every component inside the virtual machine, while the service provider manages servers and orchestrators (or containers). Examples of IaaS include Amazon Web Service Elastic Compute Cloud, Google Compute Engine, etc.

Table 2.1: Resource ownership comparison.

	On-Site	IaaS	PaaS	SaaS
Application	●	●	●	○
Data	●	●	●	○
Runtime	●	●	○	○
Middleware	●	●	○	○
OS	●	●	○	○
Virtualisation	●	○	○	○
Networking	●	○	○	○
Server	●	○	○	○
Storage	●	○	○	○

*managed by ● the user, ○ the cloud provider

2.1.2 IaaS Security Considerations

As shown in Table 2.1, IaaS users remain in control of their entire virtual environment. Users can interact with the instance as with any other machine (e.g. root access). They also benefit from several web-based application programming interfaces (APIs) that enable managing instances (e.g. selecting a hardware configuration). Therefore, IaaS minimises the trust that needs to be extended to the cloud provider. The user is in control of the (sensitive) data being processed in the virtual machine, including data from any other service built upon it (e.g. platform or software). This suggests that IaaS instances represent the most compelling targets to malicious individuals. An attacker which gains access to an IaaS instance can potentially compromise the entire OS-to-application stack.

Cloud models can be further classified depending on whether the base infrastructure is managed privately or by a third-party. In a public cloud solution, the cloud provider supplies the servers, network, and virtualisation support. The user looses physical control over its data, and the computing environment is somewhat shared with other parties. In a private cloud solution, the base infrastructure is managed locally (e.g. intranet) and can be closed from the public. It implies a significant investment on behalf of

the organisation, which might not always be justified. This solution is preferred when compliance against high security standards is required. Private clouds will either be physically disconnected from the Internet, or put under a strong firewall. These can still be targeted, however public instances certainly represent a lower-hanging fruit for the attacker.

2.1.3 IaaS Industry Outlook

Cloud providers must deploy tremendous resources in order to cope with the existing demand. Thus the market of public IaaS is dominated by a few large corporations including Amazon Web Service, Microsoft Azure, Alibaba Cloud, Google Compute Engine, Tencent Cloud, and Huawei Cloud. Table 2.2 shows that in 2019 [47], Amazon holds the largest share of the market (45%) with a revenue of 19990 million USD. In 2020 [48], Amazon still holds the largest share (40.8%) with a revenue of 26201 million USD. It can be seen that most cloud providers increase their market share over time, at the exception of Amazon which lost 7.1% between 2019 and 2020. It is also seen that the China-based cloud provider Huawei overtook Tencent in 2020. The most significant observation is the decrease of "other" cloud providers' share, indicating that it will become harder and harder to compete with the top five IaaS vendors.

Table 2.2: Worldwide public IaaS share and revenue (million USD) [47, 48].

Cloud provider	2018		2019		2020	
	Revenue	Share	Revenue	Share	Revenue	Share
Amazon	15495	47.9%	19990	45.0%	26201	40.8%
Microsoft	5037.8	15.6%	7946.6	17.9%	12658	19.7%
Alibaba	2499.3	7.70%	4060.0	9.10%	6117.0	9.50%
Google	1313.8	4.10%	2365.5	5.30%	3932.0	6.10%
Tencent	611.80	1.90%	1232.9	2.80%	-	-
Huawei	-	-	882.00	1.90%	2672.0	4.20%
Others	7425.0	22.8%	8858.0	19.9%	12706	19.8%
Total	32382	100%	44453	100%	64286	100%

In a separate study [46], the forecast global revenue is expected to reach 50393 million USD by the end of 2020, 64294 million USD by the end 2021, and 80980 million USD by the end of 2022. Between 2019 and 2022, this represents a 82% increase, which confirms the growing interest of organisations for cloud solutions.

2.1.4 IaaS Orchestration

Cloud orchestration consists of automating the dynamic management of workloads such as assigning storage and processing resource, and instantiating VMs. Orchestration also enables enforcing information flow policies such that permissions to connect and execute workloads are securely handled. Orchestration thus enables the provider to cope with an elastic demand while maintaining logical isolation among tenants. However, the isolation is not necessarily physical. Instances may be mapped to resource within the same processor, same rack, or same cluster of servers [124, 60, 167]. This is referred to as *co-tenancy* and is transparent to the cloud service user. A tenant is unaware of the activity of co-resident workloads and vice versa. This can cause undesirable effects such as "noise" generated by resource-intensive workloads on neighbour instances. A more serious consequence is the absence of barriers that would prevent a malicious cloud user from spying on its neighbours through the monitoring of the availability of shared hardware resource, i.e. microarchitectural states. Co-tenancy, or co-residency, is a core assumption to the adversary model of every microarchitectural covert and side channel attacks designed for multi-tenant computing environments. We note that cloud service providers propose products known as dedicated instances [32, 134], where the cloud user is guaranteed to be the sole tenant of a hardware platform. It is not known though at which level the physical separation is made (core, processor, NUMA node, etc.). Furthermore, this type of instance is significantly more expensive than non-dedicated instances, e.g. an on-demand EC2 a1.2xlarge instance costs 0.204 USD per hour while a dedicated EC2 a1.2xlarge instance costs 2.2162 USD per hour (at the time of writing).

Depending on the locality between a spy and a trojan, three scenarios exist for establishing microarchitectural covert channels. Let us define a communicating entity (i.e. either trojan or spy) as a process. First, if both processes are executed within the same processor core, these will share core-level microarchitectural resource including level 1 instruction and data caches, execution units, and CPU-level buffers. In fact, the two processes will share every possible components, both inside and outside of the processor core. This scenario is only plausible if SMT is enabled which allows multiple processes to share a single physical processor core. The first generation of microarchitectural covert and side channel attacks made such assumptions [2, 116, 142], which led industry and academia to devise core-level countermeasures such as *RPcache* [82], hardware cache partitions [37], noise injection [158], or even advocate the disabling of SMT [98]. Second, if both processes execute on separate processor cores but within the same processor die, these will no longer share core-level resource but only processor-level components including the last-level cache. This has been the target of the second generation of microarchitectural timing channels [171, 56, 92], leading to a new set of countermeasures being proposed including cache colouring [33], noise injection on timers [99], and recommending the disabling of memory de-duplication (which only deters the FLUSH+RELOAD class of attacks). Finally, if processes are executed on separate processors within a NUMA configuration (i.e. connected by a bus interconnect), core-level resource can still be manipulated although the non-uniform nature of memory requests may prevent accurate resolution of the leaked information. Other resource located beyond processors, such as the interconnect and DRAM row-buffers have been exploited to demonstrate new attack vectors [122, 133, 132]. Potential countermeasures include the partitioning of the memory controller [154], auditing [176], and forcing privileges on flushing instructions [56].

Achieving co-residency is further discussed in Chapter 3. The remainder of the thesis consists in identifying means of exposing that activity of co-tenants, and using it to craft rogue communication channels that circumvent logical barriers.

2.2 Computer Architecture

This section presents microarchitectural elements involved in routine processor operations and which can be exploited for timing-based covert channels.

2.2.1 Processor Core

The term *processor* refers here to the entire die. The processor comprises a last-level cache (LLC) shared among one or more cores, an integrated memory controller, buses, and additional analog components. Each core contains individual instruction (L1-I) and data (L1-D) level 1 caches, and potentially a level 2 (L2) cache depending on the microarchitecture. Cores also contain one or two physical CPU(s) each. The CPU is the set of execution units and other logic required for instruction execution, e.g. arithmetic logic unit (ALU), floating point unit (FPU), load-store unit (LSU), translation lookaside buffer (TLB), return-stack buffer (RSB), branch target buffer (BTB), etc. The execution pipeline involves resources of the CPU as well as L1 and L2 caches. A generic RISC pipeline can be decomposed in five stages, namely instruction fetch, instruction decode, execution, memory access, and write back. In microarchitectures that rely on a x86 instruction set architecture (ISA), execution stages can be different due the varying length of CISC instructions, although the overall principle remains similar. ISA implementations vary significantly from one another, however the reader will find that the above-described components are commonly encountered on modern processors (i.e. found in commercial workstations or servers). ISAs then feature many extensions to provide support for specific operations, including advanced vector extension (AVX), streaming single instruction multiple data (SIMD) extensions (SSE), instructions for optimising AES operations, or transactional synchronization extensions (TSX).

A program is a sequence of instructions which are stored into memory upon execution. The cache is there to decrease the latency of loading repeated instructions and data structures. A dedicated register, known as the program counter (PC), holds the address of the next instruction to be fetched. Unless the program encounters branches, instructions are fetched

sequentially by simple increment of the PC, e.g. plus 4 bytes on a 32-bit ISA. Instructions contain several fields, namely its opcode and operand(s). After being fetched, the instruction's opcode is decoded, thus informing the CPU on how to read the remaining of the instruction and what resources must be allocated, e.g. ALU, LSU, etc. Then, the instruction operands are propagated through the execution units. For instance, the ALU will perform arithmetic or bitwise operations on integer binary numbers, the FPU will compute more complex operations (e.g. square root) on floating-point numbers, etc. Not all instructions are of arithmetic nature. The LSU queues and executes load/store operations occurring between registers and other memory subsystems (e.g. caches). Finally, the result of the execution is written back to registers.

2.2.2 Caches

A cache is a static RAM (SRAM) type of memory. SRAM cells are made of multiple transistors, and are characterised by their ability to perform very fast read and write operations. They act as buffers for frequently requested data structures, thus improving the overall throughput. Caches are organised in several levels, with the lower levels being the closest from execution units (e.g. L1). The L1 cache is further split into an instruction cache (L1-I) and a data cache (L1-D). Other cache levels are unified, meaning they can contain both data and instruction. Other small caches may serve specific roles, such as storing page-table entries for the memory management unit (MMU), i.e. the TLB.

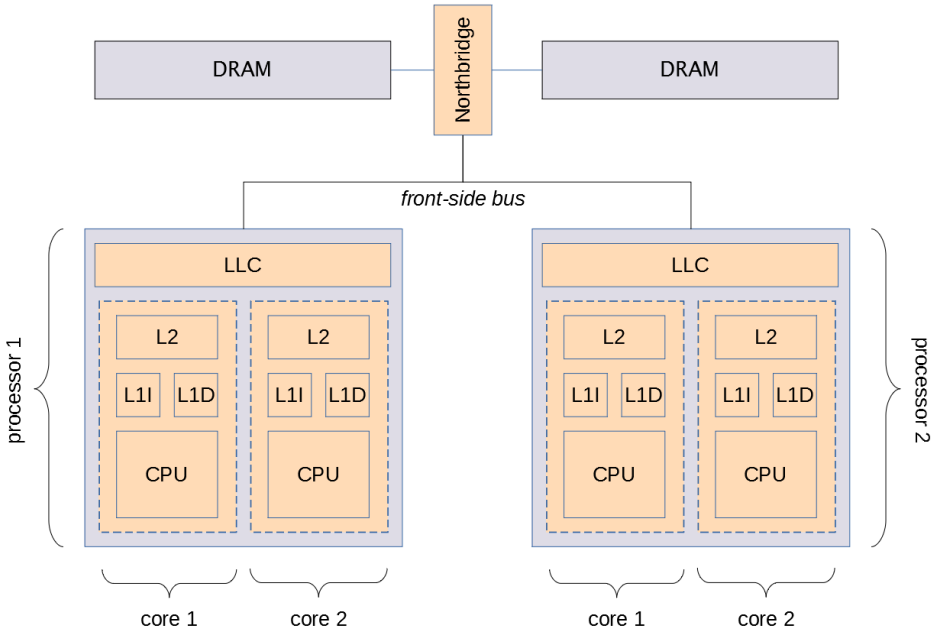
Caches can be directly-mapped, set-associative, or fully-associative. In a directly-mapped cache, a cache line index is used to determine the cache line of interest, and a tag determines whether the cache line contains the memory address. The index is a n-bit portion of the address and is therefore not unique. In other words, directly-mapped caches might contain many cache lines with the same index, also known as congruent, resulting in frequent cache line evictions. Set-associative caches somewhat mitigate the congruency issue by mapping the index to a cache set rather than a cache line. The

cache set itself contains multiple cache ways which can be differentiated with the tag. On contemporary processors, most of data and instruction caches are set-associative. Finally, fully-associative caches use the tag to directly resolve the cache way of interest, much like if a set-associative cache had only one large cache set. The TLB is often a fully-associative cache.

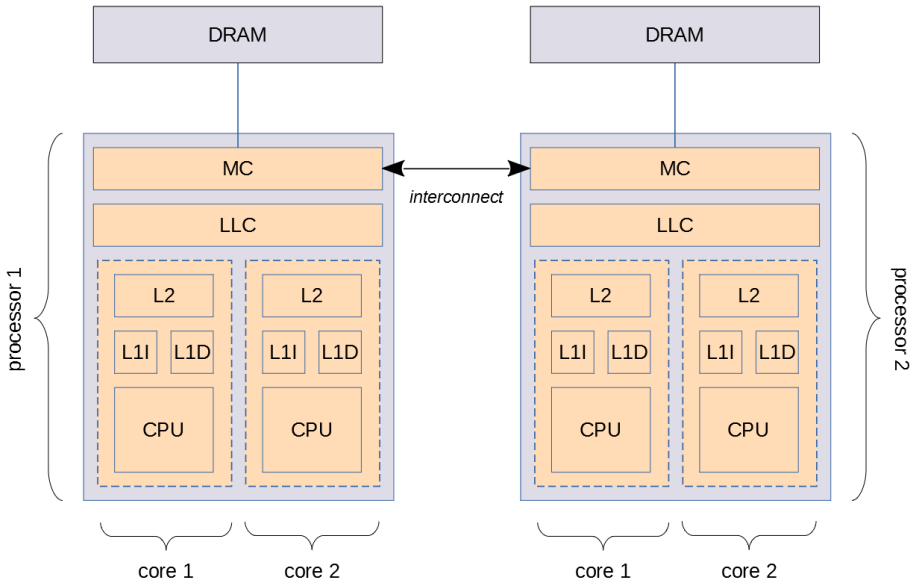
Caches also vary depending on their addressing mode. Modern processors allow multiple processes to execute concurrently, with each process having its own (virtualised) address space. As mentioned earlier, set-associative caches rely on a tag and an index in order to address memory. Therefore, the question is whether these two elements are calculated from the virtual (contiguous) or the physical (non-contiguous) address. The virtually-indexed virtually-tagged addressing mode has the benefit that it is faster since it does not require virtual-to-physical address translation. However, it requires the entire process' set of cache lines to be invalidated upon context switches (i.e. when another process takes over the execution pipeline). This is particularly expensive for large caches. The physically-indexed physically-tagged addressing mode suffers from the opposite pros and cons. A sensible approach is to rely on a virtually-indexed physically-tagged addressing mode which uses the virtual address to compute the index, and the physical address to calculate the tag. It enables retrieving the cache set immediately, thus masking the latency of translating the virtual address.

2.2.3 Memory Bus

Up until the release of the AMD Opteron and Intel Nehalem microarchitectures, memory accesses to non-cache memory were handled by a chipset composed of a Northbridge and a Southbridge. The Northbridge contained the memory controller and was linked to processors by the front side bus (FSB) (see Figure 2.1a). The Southbridge contained the I/O controller and was connected to other peripherals via dedicated buses, e.g. PCIe, SATA, etc. With the advent of parallel computing, the FSB quickly became a bottleneck for parallel program execution. An optimisation strategy has consisted in placing memory controllers directly onto processor dies. As a result, each



(a) Front-side bus model.



(b) Non-uniform memory access model.

CPU has been given fast access to a segment of DRAM, without relying on a shared memory bus. An interconnect between processors has also been introduced, in order to provide access to non-local regions of the DRAM. Accesses to local and non-local segments vary in latency, hence this architecture was named non-uniform memory access (NUMA) (see Figure 2.1b). The interconnect technology varies across microarchitectures and manufacturers. It is known as Quick Path Interconnect on Intel processors and HyperTransport on AMD processors. The Southbridge, also known as I/O hub, remains on NUMA architecture.

2.2.4 DRAM Organisation

Because cache memories are limited in size due to SRAM cells' higher footprint, memory accesses will often be served from the dynamic RAM (DRAM). A DRAM cell consists of one transistor and one capacitor. In order to read a bit, the *word line* is raised, enabling the capacitor to discharge in the *bit line*. In order to write a bit, the bit line is set accordingly to the bit value and the word line is raised long enough to either drain or charge the capacitor. Reading a bit effectively discharges the cell, and the capacitor naturally loses its charge over time. Consequently, cells need to be recharged at regular intervals. The memory controller automatically performs this operation, which can cause the read/write bandwidth to drop occasionally. Furthermore, the time required to charge or discharge the capacitor causes usual read and write operations to take longer than with SRAM cells.

DRAM cells are organised in rows, columns, banks, ranks, and dual-inline memory modules (DIMMs). Typically, a DDR3 DIMM contains two ranks, each one containing four to eight banks. Figure 2.2 represents several banks within a rank. Each bank contains a row-buffer, and a 2-D array, i.e. rows and columns. When the requested data is contained in the row-buffer, it is a *row-hit*, otherwise it is a *row-miss*. Upon row-misses, the row-buffer is updated with the row containing the requested data, before serving the request. As a result, a row-miss has a significantly higher latency than a row-hit.

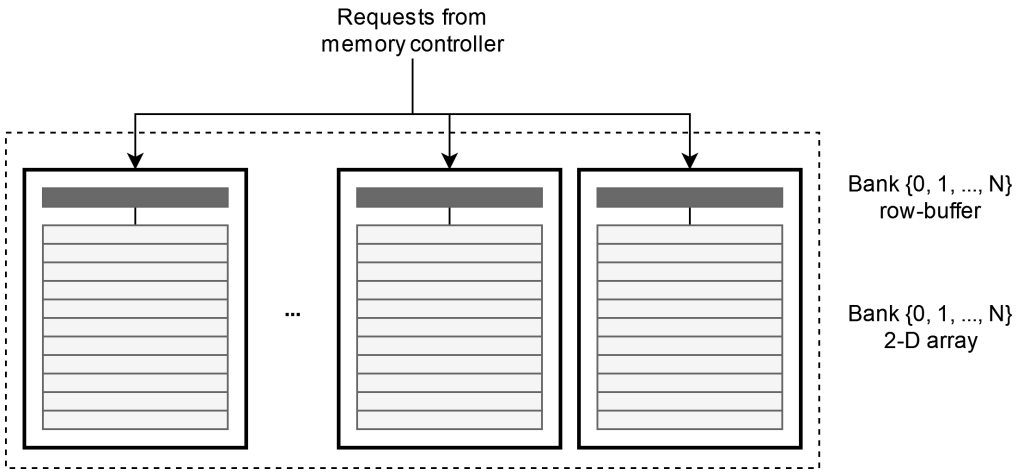


Figure 2.2: DRAM architecture.

2.2.5 Memory Controller

The integrated memory controller, also known as DRAM controller, contains storage and scheduling resources to arbitrate memory accesses. The request is first stored in the buffer matching the DRAM bank that it targets. Then, the bank scheduler prioritises requests according to a scheduling algorithm. Once a request wins bank arbitration, it is rescheduled by a channel scheduler. Again, the scheduling algorithm determines priorities. Usually, requests that target open-pages are served first, so as to mitigate the latency incurred by updating a row-buffer. Before serving the memory access, the memory controller must translate the requested data's physical address into a DRAM map (i.e. channel, rank, bank, row, and column). The physical-to-DRAM address translation is performed according to DRAM addressing functions.

The memory controller's page policy dictates the aliveness of data in the row-buffer. If a close-page policy is enforced, the row-buffer will systematically be cleared after serving a request. Thus, each memory access results in a row-miss, preventing timing variations, but globally slowing down the execution of programs. If an open-page policy is enforced, the row-buffer will retain data until it must be updated with a new row. Thus, it allows the occurrence of row-hits, reducing the global execution time of programs, but introducing exploitable timing variations.

2.3 Operating Systems

This section introduces the concept of memory virtualisation and presents the effects of memory operations on microarchitectural elements.

2.3.1 Logical Boundaries

Virtualisation of the address space allows the operating system to set access rights and privileges to memory regions, such that two processes cannot have contiguous physical memory. Virtualisation is thus a basic principle of computer security as it enables memory isolation. When a memory operation is performed by one process, the MMU ensures that the targeted region belongs to this process. In other words, a process can never access the address space of other processes running concurrently. In a virtual machine, it is the entire guest's address space that is virtualised. This guarantees logical isolation between the address space of each VM, making possible the sharing of a computing environment while maintaining separate memory spaces. It is worth mentioning that operating systems can feature *shared memory* which occasionally allows a region of physical memory to be shared. This is used to limit redundancies and reduce the global memory footprint, e.g. shared libraries.

An orthogonal concept is the segregation between the user space and the kernel space. A process running in user mode can never be able to access a region of kernel memory. In contrast, a process running in kernel mode (e.g. driver, kernel module, etc.) has unrestricted access to physical memory, including that of user processes. When a user process attempts to access kernel memory, it is an illegal memory access. Kernel privileges should not be mistaken with superuser or administrative privileges. A process running with superuser permissions only has access to the kernel mode features that the kernel exposes to it (e.g. accessing system files). However, it is still a user process, and as such it does not have unrestricted access to peripherals and physical memory. Further modes of operation exist depending on the processor ISA. Seeking to obtain kernel privileges from an unprivileged program is referred to as a privilege escalation attack.

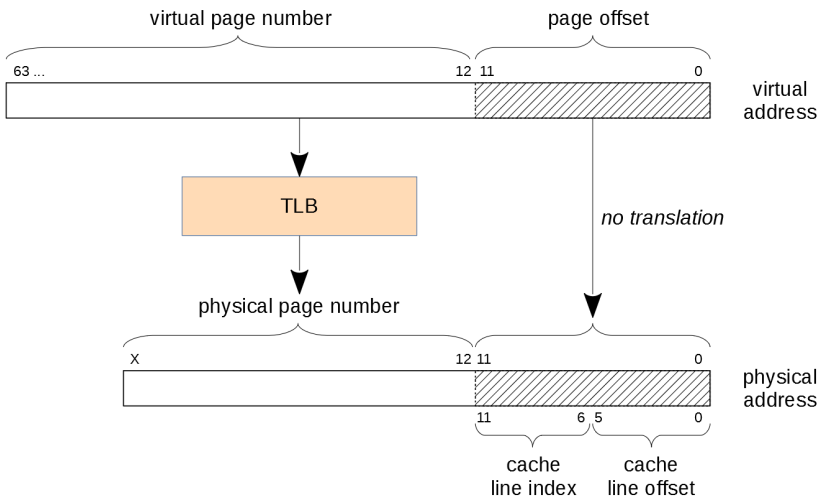


Figure 2.3: Virtual-to-physical address translation.

2.3.2 Virtual-to-Physical Address Translation

A process running on a processor that employs an addressing mode with virtual indices manipulates data structures using virtual addresses. When a data structure must be read or written to, a memory access occurs. This requires the virtual address of the data to be translated into a physical address. The MMU features its own cache, known as the TLB. The TLB contains the virtual-to-physical map of recently accessed addresses. If the TLB already contains the required entry, it is known as a *TLB-hit*, otherwise it is a *TLB-miss*. Upon TLB-misses, the MMU performs a page table-walk to recover the translation map. Once the physical address has been retrieved, the MMU computes the index and the tag of the cache line that contains the requested data. The index is used to point to a set of cache lines, and the tag is used to point to a specific cache line within this set. Finally, an offset computed from the variable's physical address is used to point to a specific portion of the cache line. Figure 2.3 shows how virtual-to-physical address translation is performed. The virtual page number is mapped to a physical page number via the TLB, while the 12-bit page offset is unmodified. Furthermore, the 6-bit cache line offset indicates a cache line size of $2^6 = 64$ B, and the cache line index indicates a way size of $2^6 = 64$ entries.

2.3.3 Memory Access and Latency

Once that the physical address has been recovered, the access to memory can proceed. If the requested cache line is not present at any cache level, known as a *cache miss*, a request is issued to the memory controller in order to fetch the data from DRAM. The data is then stored into the cache, known as a *cache line fill*, and is sent back to the CPU. The next access to the cached data will result in a *cache hit*. A store operation consists of modifying a cache line, and storing it back to memory (depending on the write-policy) via the store buffer. Prior to modifying data, the cache line must be loaded. If it is not present at any cache level, it is called a *write miss*, which triggers a cache line fill (except for Pentium processors). Otherwise it is a *write hit*.

In terms of latency, a TLB-miss will cause a longer access than a TLB-hit. Similarly, a cache-miss will serve the requested data more slowly than a cache-hit. Finally, if the data is served from DRAM (i.e. upon cache-miss), the row-buffer will also influence the latency, i.e. a row-miss causes a row-buffer update to occur before serving the memory access and is therefore slower than a row-miss. If the requested data is still not present in DRAM, it will be fetched from persistent memory, e.g. the disk. Accesses to the disk tend to have a dramatic effect on performance.

2.4 Related Work

This section presents the related in the field of microarchitectural attacks, from transient attacks to timing-based covert channels.

2.4.1 Microarchitectural Attacks

The field of microarchitectural attacks encompasses many sub-categories. Due to the large amount of publications, researchers attempted to classify and occasionally draw taxonomies of attack techniques [7, 13, 20, 21, 50, 75, 82, 173, 95, 141]. Certain surveys focus exclusively on vulnerabilities in cryptographic algorithm implementations [15, 38, 42, 43, 94], while others

examine the effects on trusted execution environments [28, 103, 127]. In the following sections, an overview of the different sub-categories is presented.

Transient attacks

Transient-execution attacks are those that exploit hardware optimisation features such as out-of-order, speculative execution, etc. Rather than executing instruction sequentially, modern processors have the ability to re-order micro-operations (μOPs) so as to optimise resource utilisation. For example, if an μOP requires an execution unit that is not available, it may be postponed such that other μOPs can be treated. Once the unit becomes available, this first μOP is executed, and the results are committed in-order. Occasionally, it is the evaluation of a condition for branching that is executed out-of-order. In such cases, the CPU can bet on the outcome of the condition evaluation, and speculatively execute a set of instructions. Whenever the condition is actually evaluated, if the bet was incorrect, than the CPU rolls back into its previous state.

The Meltdown class of attacks [26, 89, 126, 146, 147] abuses out-of-order execution in order to access kernel memory from a userspace program. The attacker performs an access to a virtual address that maps to kernel memory, causing the OS to raise an exception. To prevent program termination, Lipp et al. [89] suggested to either implement the illegal access into a child process, or use Intel TSX [66] to roll back into a previous state. The transient instruction, i.e. the one executed out-of-order but whose results were never committed, loaded the content of the virtual address into cache memory. The attacker then launches a cache-based covert channel in order to recover the contents of the kernel memory.

The Spectre class of attacks [30, 81, 83, 97] instead leverages speculative execution, in order to access concurrent programs' memory. The attacker mistrains the BTB by repeatedly executing a conditional branch with a valid condition. Once that the BTB is biased, the attacker can trick the CPU into executing speculatively a set of instructions which have a dependency on the targeted secret. To do so, the evaluation of the conditional branch must

depend on resources that are not immediately available, while the secret shall depend on resources that are already available (e.g. if `secret` = `array`[`x`], then `array` and `x` must be cached). Finally, the attacker launches a covert channel to recover the secret located in cache memory. Other variants of the Spectre attack have later been developed with different means of extracting the secret data, e.g. through a gadget that can be executed by mistraining the BTB (SpectreV2), using the Meltdown attack principle (SpectreV3), or re-using other covert channel primitives (SpectreV4).

Timing-based attacks

Timing-based microarchitectural attacks leverage the sharing of hardware resources in order to steal and/or transmit information illegitimately. For instance, several works showed that a hardware thread may spy on the activity of another thread by monitoring the availability of execution units and their associated buffers [3, 4, 12, 25, 40, 157]. Similarly, it was also demonstrated that the TLB [63] and core-level caches (e.g. L1-I) can leak information at runtime [1, 2, 151, 160, 179]. A trivial countermeasure to these attacks consist in segregating threads belonging to different security domains onto separate cores. As a result, attackers moved towards hardware resources that are shared among all processor cores. The LLC is one of such resources, and has enabled a plethora of new cache attacks on both x86 [10, 27, 56, 180, 76, 169] and ARM [88, 174, 177] platforms.

Among these cache attacks, the most eminent one is the Prime+Probe [116] technique: the attacker fills one or more sets of a set-associative cache with its own cache lines (i.e. priming), lets the victim execute, and measures the access time to the data stored in these cache lines (i.e. probing). Whenever an access to a cache line is slow (i.e. cache-miss), it indicates that the victim evicted the data due to its own operation, revealing the presence of a shared cache set. Another prominent cache attack is the Flush+Reload [171] technique: the attacker flushes a cache line of shared memory with victim, lets the victim execute, and reloads the same cache line. If the reload is fast (i.e. cache-hit), then it indicates that the shared

cache line has been loaded by the victim. While Flush+Reload relies on the availability of shared memory, it does benefit from a higher resolution than Prime+Probe, i.e. the size of a cache line rather than the size of a cache set.

Osvik et al. [116] also presented the Evict+Time technique: the attacker lets the victim execute, causing the cache to fill up, and performs eviction with its own cache lines. Then again, the attacker lets the victim execute, and probes access to its own cache lines. If the latter is slower than normal, then indicates that the victim accessed the cache line of interest. Evict+Time is less encountered in the literature, as it does not provide any benefit over its counterpart. Gruss et al. developed the Flush+Flush technique where the attacker measures the elapsed time of each flushing instruction. If the second one has the same latency as the first one, it indicates that the victim performed a cache line fill in between, revealing the victim's data structures at the cache line granularity.

Timing attacks have found multiple applications. For instance, a large body of research has focused on finding vulnerabilities in Intel's Software Guard Extension (SGX). SGX is a proprietary TEE which enables an application running securely within a non-trusted environment. While Intel specifies that its TEE is not designed to safeguard against timing-based attacks, it does present salient flaws related to its caching [36, 54, 103, 104, 129, 153], page-table [59, 148, 153, 165], branch prediction [41, 86], and DRAM addressing [153] mechanisms. Naturally, most of timing-based side channels have also been applied against cryptographic implementations. The amount of related attacks does demonstrate how approved algorithms remain vulnerable under flawed implementations: (T)DES [143, 144], ECC [18, 24, 149, 170], RSA [5, 65, 172], and AES [6, 19, 22, 57, 72, 73, 111, 142, 159]. Nowadays, the literature focuses more on complex mechanisms such as TEEs rather than cryptographic libraries.

Focus on timing-based covert channel attacks

Microarchitectural cross-VM covert channels are software-launched attacks which exploit multi-tenant environments' shared hardware. In a timing-based covert channel, variations in the latency of a program execution are used to encode binary information. The receiver probes its own execution, which is directly influenced by the state of a hardware resource shared with the sender. An entire stream of bits is reconstructed in this way, enabling an attacker to transmit information from a compromised VM to another co-located VM. Ristenpart et al. [124] first studied the problem of VM co-location on the AWS EC2 service. They used the LLC to assert of the co-residency between two communicating VMs. Similarly, Xu et al. [166] explored the vulnerability of L2 caches for covert channel attacks on an EC2 instance. Wu et al. [164] proposed exploiting the memory bus as an alternative to cache-based covert channels, thus overcoming the addressing uncertainty. Later, the memory bus attack was revisited by Liu et al. [93] to use non-temporal instructions on the receiving-end, so as to mitigate the effect of cache pollution. This technique was previously suggested by Guri et al. [58] in their air-gapped covert channel. Pessl et al. [122] suggested using the DRAM row-buffer as a communication medium between two VMs. Their attack also allows cross-processor information leakage. Liu et al. [92] re-used a Prime+Probe primitive in order to build a cross-VM covert channel as a vector for side channel attacks against GnuPG libraries. Maurice et al. [101] designed a robust LLC-based covert channel attack. Their work demonstrated the feasibility of implementing a network protocol on top of a covert channel across AWS EC2 instances. Sullivan et al. [140] revisited the exploitation of SMT using the memory order buffer for cross-VM leakage in the AWS EC2 and Google Cloud Engine (GCE) services. Their experiment highlights the consequence of enabling hyperthreads on public cloud platforms. Schwarz et al. [128] suggested exploiting the latency of powering up the upper-half of the AVX2 unit in order to build a new covert channel, subsequently used in their remote Spectre attack [81].

2.4.2 Countermeasures

This section surveys and analyses relevant mitigation techniques against microarchitectural leakage channels, namely noise injection, software partitioning, and hardware partitioning. Whether a timing variation is created accidentally or intentionally, the mechanisms to modulate microarchitectural states remains similar. Therefore relevant countermeasures against timing-based side channels are also considered. Other countermeasures which are not relevant include constant-time execution, symbolic execution, state flushing, and noise injection within cryptographic implementations.

Noise injection on timers

This approach consists of jittering the timestamps of high-resolution timers [62, 152, 99, 93]. Being able to measure the latency of a single memory operation is crucial in timing channel attacks, as it leads to the interpretation of the activity of the victim (or sender). The x86 ISA features the `rdtsc` and `rdtscp` instructions which capture a time-stamp from the time-stamp counter (TSC), allowing timing measurements with a sub-nanosecond resolution. These are accessible from any non-privileged user program. Other timing sources, such as the wall clock provided by the operating system, are usually not accurate enough to measure a timing variation of a few clock cycles. For example, in [133], the sender's activity generates an overhead of only 6.5 CPU clock cycles. At a frequency of 2.4 GHz, this amounts to a time span of 2.7 ns. The attacker can neither rely on high-resolution timers, nor on operating system wall-clocks

Noise injection on caches

This approach aims at preventing an attacker from learning about the victim's working cache set. Wang and Lee [158] suggested integrating permutations in the cache index computation, while Qureshi et al. [123] used randomised mappings based on the encryption of the cache line's physical address. These will result in the victim's accesses to stop conflicting with the attacker's cache sets. Alternatively, Fang et al. [44] suggested having

the prefetch controller issuing requests to the L1 cache in order to tamper the timing observations of the receiving-end. For instance, in an m-way set associative cache, if m cache misses are observed when sending a 1, and none are observed when sending a 0. the prefetch controller will bring this number to $m/2$ all the time, such that the receiver is no longer capable of distinguishing a 1 from a 0. If generalised, these strategies can hinder cache-based covert channels that depend on the ability to find congruent addresses. Other proposals [91, 45, 168] studied bespoke cache replacement policies as an alternative to the vulnerable on-demand policy. Taking the example of the random-fill approach [91], if a cache miss occurs, the requested cache line is sent to the CPU but it is not necessarily stored in the cache. Instead, a "neighbour" cache line is randomly selected within a fixed address range around the requested cache line. If the same cache line is requested thereafter, it might result in a cache hit. The uncertainty contributes to inhibiting the leakage of information as to whether the victim accessed a specific cache line or not. This countermeasure is also relevant to cache covert channels such as Flush+Reload [171].

Software partitioning

Software cache partitioning, also known as cache colouring, consists of isolating sensitive data by means of isolating a set of cache lines for a given security domain [158, 155, 78]. Recall that in order to address data in (set-associative) caches, the MMU computes an index and an offset from the physical address. The bits that belong to both the physical page number and the cache line index are the *colour bits*. Figure 2.4 is an example of virtual-to-physical translation of a 64-bit address, with 6 bits of offset (i.e. cache line size is $2^6 = 64$ bytes), 9 bits of index (i.e. way size is $2^9 = 512$ entries), and 3 colour bits. Cache colouring states that physical pages which differ in any of the colour bits can never be mapped in the same cache set. That is, if the physical memory pages of two processes have at least one different colour bit, these can never exploit congruency to launch cache attacks such as Prime+Probe or Evict+Reload [116]. In a sense, cache colour-

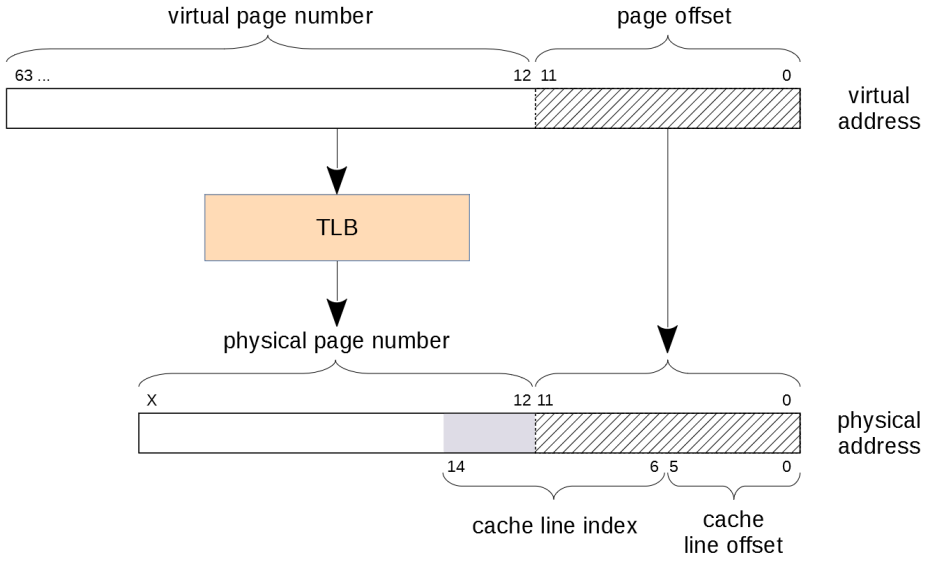


Figure 2.4: Virtual-to-physical translation of a 64-bit address on an Intel E6550 processor [33]. The cache line offset is determined by bits 0 to 5, the cache line index is determined by bits 6 to 14, and the page offset is determined by bits 0 to 12. Colour bits range from bit 12 to bit 14.

ing behaves like a dynamic clustering technique which guarantees that two clusters can never share a cache set. Liu et al. [90] suggested another form of software cache partitioning by leveraging Intel's Cache Allocation Technology (CAT) [69], in order to lock down portions of the LLC during execution. As for Flush+Reload, Zhou et al. [181] proposed a state machine which prevents a shared memory page being accessed by two security domains at the same time.

Beyond cache colouring, other forms of software partitioning have been proposed. Disabling page sharing [98] hinders attacks which depend on the availability of shared memory such as Flush+Reload and Flush+Flush [56]. Disabling SMT [17] prevents two hardware threads from exploiting contention among CPU-level resources such as execution units [12], the BTB [4], the RSB [25], or the MOB [140].

Hardware partitioning

Hardware cache partitioning consists in providing physical isolation among the working cache sets of each tenant [158, 118, 37]. For example, Wang and Lee [158] suggested a cache line locking mechanism, by means of an ISA extension, which prevents another process from evicting the cache line. An L tag indicates whether the cache line is locked, and an ID tag indicates the process to whom the cache line belongs. Fundamentally, hardware cache partitioning results in the same effects as software cache partitioning. The main difference lies in the deployment of the countermeasure. Therefore, hardware cache partitioning does not result in additional requirements. As for other components than caches, Wang et al. [154] proposed a time-division multiplexing technique in order to prevent the exploitation of the shared integrated memory controller. Similarly, Wang et al. [156] devised a priority-based mechanism for the shared on-chip network. These approaches consist in scheduling accesses to the memory controller and the interconnect such that different security domains cannot conflict with each other. The effect of this countermeasure on DRAM-based covert channels that target external NUMA nodes remains an open-question.

Additionally, Gruss et al. [56] advocated making the `rdtsc` and `clflush` instructions privileged. While it would not completely close the covert channels which rely on these instructions, it would severely question the practicality of the attack. The adversary model (see Section 5.3.2) requires for the environment of the victim to be compromised with a malicious colluding software. The above-mentioned countermeasure would force this malware to be executing with privileges.

Chapter 3

On the Severity of Covert Channel Attacks in IaaS

Contents

A question that systematically arises from the audience when presenting a new leakage channel is "How practical is this attack?". Microarchitectural covert channels allegedly rely on a strong adversary model. As a result, security experts from industry tend to overlook malicious covert channels. This chapter intends to clarify the operational constraints associated to the deployment of such attacks in IaaS environments. Furthermore, a dedicated scoring framework that integrates operational constraints in the covert channel's performance score is devised.

3.1 Introduction

Cross-VM microarchitectural covert channel attacks always consider two processes, a *trojan* and a *spy*, who want to share information illegitimately. The trojan possesses sensitive information and intends to transmit this information to the spy. The two processes run in separate virtual machines and thus have dissociated address spaces. The security policy forbids these two entities from communicating directly, as it would be the case in any IaaS environment (see Figure 3.1). The threat model assumes that the instance of the victim has been compromised with the trojan. This is anything but an unrealistic scenario. Indeed, software supply chain attacks such as SolarWinds [120] have been increasingly observed in the wild [114]. These are deployed through corrupted package managers and tend to remain undetected for long periods of time. The threat model also assumes that the attacker achieved VM co-location between the two VMs, i.e. they are running on the same bare-metal hardware. Several research efforts have demonstrated that this can be achieved on public cloud infrastructures [60, 117, 124, 167], as discussed in Section 3.3.2. The remaining of the threat model (i.e. privilege level, hardware locality, dual communication, shared memory, etc.) is specific to the attack.

In this chapter, a measurement study on the practicality and severity of deploying microarchitectural covert channels in IaaS is conducted, based on the latest version of the CVSS (v3.1) [113]. The CVSS is an open industry standard that is widely used in the security community in order to assist responses to threats. Other evaluation frameworks exist [74, 139, 138, 77], however the CVSS remains the established industry standard for rating an attack's severity. Its evaluation criteria are discussed in the context of microarchitectural attacks, and state-of-the-art covert channels previously surveyed (see Section 3.2) are analysed.

Although comprehensive and well-established, the CVSS is designed to evaluate a large range of vulnerabilities. As such, it may struggle to outline disparities among different attacks in an area as specific as microarchitectural covert channels. In order to provide a fair and realistic point of com-

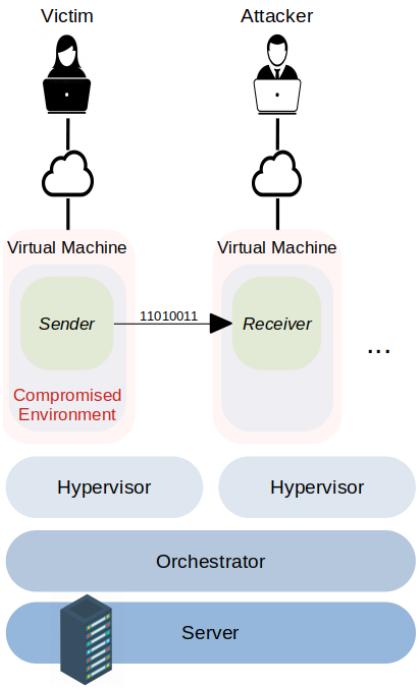


Figure 3.1: Attack setting.

parison, a new dedicated evaluation framework is proposed, i.e. the Covert Channel Scoring System. The CCSS outlines the effect of operational constraints on the severity score by accounting for requirements over privilege levels, hardware locality, initialisation, existing countermeasures, and effective communication speed. Among other findings, it shows that cache-based covert channels achieve the lowest severity scores, despite being increasingly popular. The work presented in this chapter is based on the publication [131].

3.1.1 Scope of the Study

In addition to the definitions provided in Section 1.4, the scope of this study is further defined here. A covert channel does not necessarily allow communication across VMs. However, this research thread focuses on the threat against public clouds, therefore only cross-VM covert channels are considered. The same methodology could be applied to similar environments, e.g. private clouds. This evaluation is specific to IaaS, and does not account for

variations in similar environments.

3.1.2 Structure of this Chapter

Section 3.2 provides an analysis of state-of-the-art cross-VM covert channels. The inner mechanics of each attack are summarised, and an overview of the associated operational constraints is presented.

Section 3.3 defines every criteria used in this study. Metrics listed from Section 3.3.1 to Section 3.3.10 are taken from the CVSS v3.1 [113]. These serve as input for the CVSS evaluation. Metrics listed from Section 3.3.11 to Section 3.3.13 are bespoke criteria which serve as input for the CCSS evaluation.

Section 3.4 presents the scoring equation of the CCSS. The CCSS consists of a new methodology for assessing the practicality of covert channels. Its output is discussed in Section 3.5.

Section 3.5 provides the results of both the CVSS and CCSS evaluations. It also discusses the pros and cons of each scoring system. Finally, Section 3.6 summarises the contributions of this chapter.

3.2 Analysis of State-of-the-Art Attacks

In order to evaluate microarchitectural covert channels against the CVSS criteria, it is first required to understand the inner mechanics of each attack, and how they can be mitigated by potential countermeasures. These are listed in Table 3.1.

3.2.1 Memory Order Buffer

The memory order buffer (MOB) attack [140] exploits a side-effect of write-after-read hazards named 4k-aliasing. This effect occurs whenever the lower twelve bits of the addresses contained in the load and store registers match. This causes the load operation to be re-issued, resulting in the load/store bandwidth to drop. Sullivan et al. [140] leverage 4k-aliasing in order to

create a covert communication between two hyperthreads. The sender either fills the store buffer with page-aligned addresses to transmit a one, or empties the store buffer to transmit a zero. Concurrently, the receiver probes load operations on every page-aligned addresses. When the load/store bandwidth drops, the receiver will observe a higher latency.

This effect is exploitable only at the thread-level, as it is linked to the load and store buffers located within the CPU. In other words, the communicating entities must be executed by the same physical core. To the best of our knowledge, root privileges are not required. With regard to countermeasures, authors acknowledge that disabling SMT is a straightforward way of mitigating the vulnerability. However, they also argue that "hyperthreading is expected to become more popular on IaaS platforms in the near future in order to keep them affordable". Indeed, SMT remains available on dedicated instances or for general-purpose workloads.

3.2.2 Last-Level Cache

LLC-based covert channels [124, 166, 92, 100, 101] derive from the Prime+Probe attack [116]. The receiver fills up a cache set, waits for the sender to execute, and probes its accesses to the same cache set. If the sender chose to modify the cache lines of the receiver, the latter will experience a

Table 3.1: Cross-VM covert channel attacks.

Attack	Exploited resource	Bitrate	Error	Capacity
[124]	Last-level cache	0.2 bps	-	-
[166]	Last-level cache	3.2 bps	9.28%	1.77 bps
[164]	Memory bus	343 bps	0.39%	330 bps
[92]	Last-level cache	1.2 Mbps	22%	287 kbps
[100]	Last-level cache	751 bps	5.7%	514 bps
[122]	DRAM row-buffer	596 kbps	0.4%	573 kbps
[101]	Last-level cache	45.25 kbps	0%	45.25 kbps
[140]	Memory order buffer	1.49 Mbps	~5%	1.06 Mbps
[133]	Memory controller	150 bps	7.8%	90.7 bps
[132]	Memory bus	480 bps	5.4%	333 bps

slower access. PRIME+PROBE relies on the existence of congruent addresses between the sender and receiver, i.e. virtual addresses that map to the same cache set.

Identifying congruent addresses requires performing virtual-to-physical address translation using privileged page tables. Alternatively, entities can use the page offset of *huge pages* (i.e. 2 MB) as it is not translated, and it is long enough to include index bits. The communicating entities need to agree on a set of congruent addresses, which cannot be performed in the absence of an existing communication channel. In order to cope with this issue, Maurice et al. [101] suggested using a jamming agreement. Independently of the chosen strategy, LLC-based attacks are not functional without an initialisation phase, and are limited to cross-core communication. Several cloud-oriented mitigation techniques were proposed to tackle LLC-based timing channels, such as cache partitioning or noise injection [90, 53, 78, 152].

3.2.3 DRAM Row-Buffer

The DRAM addressing covert channel [122] exploits the DRAM bank row-buffer to create timing variations on uncached memory accesses. The sender allocates memory, and performs memory accesses either in the cache or in the DRAM. When the sender accesses the DRAM, it causes the bank's row-buffer to be updated with the sender's row. Concurrently, the receiver accesses the same DRAM bank as the sender. If the sender evicted the receiver's row from the row-buffer, a row-miss occurs resulting in a higher latency.

It is trivial to extend the original author's threat model to remote and unprivileged adversaries. One entity can simply write zeroes and ones on a random memory location, and the other entity scans its memory address space to detect the bit pattern, i.e. consecutive row-hits and row-misses. This approach also enables implementing a covert channel without knowledge of the DRAM addressing function, at the cost of an initialisation phase. This covert channel has the advantage that the communicating entities do not necessarily need to be scheduled on the same processor, as the DRAM memory is shared at a system-level via the interconnect. Auditing can be

used as a mitigation strategy. The constant probing to DRAM will result in a significant amount of cache-misses, observable by hardware counters. Alternatively, the `clflush` instruction can be restricted to privileged programs, thus rendering the attack harder to implement.

3.2.4 Memory Controller

The memory controller covert channel [133], which is further described in Chapter 4, consists in modulating the load on the DRAM controller's channel scheduler in order to induce timing variations in the receiver's memory accesses to DRAM. In the cross-VM version, the sender allocates three memory pages, and then reads one byte either in each of the three pages, or in a single page. The receiver observes a higher latency when the sender is increasing the load on the channel scheduler.

This attack is feasible both with and without privileges. Communicating entities need to agree on a memory channel. As in the row-buffer attack, this can be achieved by having the sender broadcasting his position. Because the memory controller is accessible at a system level, this attack could be extended to multi-processor configurations. The memory controller covert channel can be addressed with the same countermeasures as the row-buffer one. Alternatively, the controller could be redesigned in order to enforce temporal [154] or spatial isolation.

3.2.5 Memory Bus

The memory bus covert channel, first suggested by Wu et al. [164] and later studied in Chapter 5 of this thesis, uses atomic operations on exotic memory operations, i.e. operations on cache line-crossing memory regions, in order to trigger a bus lock emulation. The sender either performs an exotic access, or remains idle. Meanwhile the receiver probes its uncached memory accesses. A high latency is observed whenever the sender accesses exotic memory regions.

This attack enables cross-core communication on NUMA architectures, and cross-processor communication on front side bus architectures. It does

not require root privileges, and remains functional in the absence of an initialisation phase. With regards to countermeasures, Chapter 5 provides an exhaustive list of potential countermeasures, however it is shown that none of them are efficient at closing the covert channel. Wu et al. [164] suggested an auditing approach where the cache-miss memory bus lock counters are monitored in order to detect performance anomalies.

3.3 Evaluation of Attacks

This section lists the criteria used to assess the impact of malicious covert channels in IaaS environments. The Common Vulnerability Scoring System is used as a base for evaluation metrics. Other frameworks could have been used. The Exploit Prediction Scoring System (EPSS) [74] is a threat-oriented approach used to estimate the probability that a vulnerability will effectively be exploited, based on reports of vulnerability disclosures and exploitation data. This framework depends heavily on the availability of datasets from intrusion detection systems, honeypots, network observatories, malware analysis, and other sensor networks. Exploitation data is currently not available for cloud-based microarchitectural covert channels. However, future developments of the CCSS shall take into account such reports in order to better understand the threat. Another alternative to the CVSS is the Stakeholder-Specific Vulnerability Categorization (SSVC) framework [139], which provides a different scoring depending on whether the stakeholder is the patch developer or the patch applier (i.e. distribution). The SSVC is motivated by the existence of diverging priorities among vendors and deployers. The former would prioritize the technical impact, while the latter would focus on mission impact. The SSVC offers a very interesting perspective to vulnerability management, and this approach could significantly contribute to enhance the CCSS (e.g. in order to assist both hardware manufacturers and cloud service providers adequatly). However, because the SSVC is in itself a refinement of the CVSS, it is currently not answering the needs of this thesis item which primarily aims at integrating evaluation metrics specific to microarchitectural covert channel attacks. Finally, there exist several legacy

frameworks such as *DREAD* [138] or *STRIDE* [77]. DREAD was found to provide inconsistent ratings and was eventually discarded by the industry in the 2000s. STRIDE is rather used to identify vulnerabilities in a system and help addressing corresponding gaps. In comparison to above-mentioned frameworks, the CVSS offers the most relevant and set of evaluation metrics for this work. The associated severity scores are presented in Section 3.5. The list of criteria is then augmented with covert channel-specific considerations, such as the hardware locality, the initialisation phase, and the covert channel capacity. A dedicated scoring framework is devised in Section 3.4, and its results are also presented in Section 3.5.

3.3.1 Attack Vector

Attack vector evaluates the proximity between the attacker and its target. It can be rated as "network" for remote interaction, "adjacent" when the attacker needs physical or logical proximity with the target (e.g. Bluetooth), "local" if it relies on user interaction (e.g. social engineering), or "physical" when physical manipulation is required.

The attack vector is rated as "local". The sending-end consists of a malicious program running inside the instance of the victim. This trojan must be inserted either using social engineering, or by corrupting the machine image. Independently of the chosen attack vector, user interaction is required.

3.3.2 Attack Complexity

Attack complexity assesses the difficulty of exploiting a vulnerability once access to the targeted platform is gained, ranked either as "low" when no specialised access condition exists, or "high" when the attack requires a significant amount of preparation such that it cannot be performed at will. The VM co-location problem is discussed under this criterion.

Attack complexity is rated as "high". Prior to launching the attack, the adversary must achieve VM co-location, independently of the covert channel considered. Cloud services' APIs do not allow an attacker to place an

instance at will on a chosen physical machine. One solution consists in using networking utilities to map the internal network topology of the data center. Ristenpart et al. [124] suggested mapping the instances' internal IP addresses to their external ones. The obtained topology allowed them to place two instances on the same processor. Herzberg et al. [60] proposed a network-based side-channel technique to deanonymise instances internal IP addresses on AWS EC2 and Rackspace Cloud services, thus improving on the previous proposal. Xu et al. [167] further studied the AWS AC2 topology, and found out that although it is has become harder to co-locate two instances (e.g. time locality vulnerability is significantly reduced), a residual threat remains. Authors were capable of achieving co-residency despite isolation countermeasures taken by the cloud provider, i.e. virtual private clouds (VPCs). Concurrently, Varadarajan et al. [150] analyzed the efficiency of VPCs against co-location attacks on AWS EC2, Google GCE, and Microsoft Azure services. They concluded that "achieving co-location is surprisingly simple and cheap". Microarchitectural covert channels can later be used to find out whether co-residency is achieved at the core-level, package-level, or system-level.

While these approaches require some knowledge of the network topology, an adversary can also choose to directly apply microarchitectural covert channels to detect co-residency. Indeed, contention generated by the sharing of hardware resource among VMs cannot be mitigated by network (i.e. logical) isolation technologies. In a purely microarchitectural co-residency attack, the sending-end can "broadcast" messages repeatedly on a covert channel, until a receiving-end picks up. Thus targeted co-residency is still possible without access to a reliable network topology of the data centre, although this methodology is more hazardous. Recently, Atya et al. [14] successfully demonstrated this approach on AWS EC2, using the memory bus and the cache as communication mediums.

3.3.3 User Interaction

User interaction indicates whether human interaction other than the adversary is required. As such, this criteria can be rated as either "none" or "required". The trojan insertion problem is discussed under this criterion.

User interaction is rated as "required". The AWS EC2 service is a practical example of means to compromise a victim's instance before its deployment. Amazon Machine Images (AMIs) are the basic unit of the EC2 service. An AMI contains the OS along with libraries, applications, and other components which personalise the instance. AMI selection presents a unique opportunity for an attacker: anyone with an AWS account can customise and share an AMI. As a result, an attacker can conceal and distribute a trojan across a large pool of users. Also, because the AMI contains a tremendous amount of code, it is extremely difficult (if not impossible) to uncover malicious code once it is embedded into the image.

Even trusted machine images, whether they belong to AWS or other cloud providers, contain software components (e.g. OS, middleware, applications, etc.) which involve a complex software supply chain. An application is rarely designed by a single company, but instead includes multiple parties. Therefore, the end-user is provided with a software application which is the result of an extremely complex supply chain. It is very difficult (if not impossible) for the user to control whether every party has applied proper security practices. The recent SolarWinds cyberattack [112] is a compelling demonstration on the exposure of the software supply chain. Whether trojan insertion is performed using social-engineering, or via machine image corruption, specific actions must be performed by the victim, hence the "required" quotation.

3.3.4 Scope

Scope assesses the impact that a vulnerability might have on components other than the one affected by the vulnerability. This metric accounts for the overall system damage caused by the exploitation of the reported vulnerability. Scope can be rated as "changed" when a scope change occurs, or "unchanged" otherwise.

Scope is rated as "unchanged". The mechanism responsible for enforcing access control over the vulnerable component, also known as the *security authority*, depends on the form of the trojan. For example, if the sending-end is part of a user application (e.g. plugin), the vulnerable component is the affected application (e.g. web-browser) and the security authority is the guest operating system, responsible for enforcing isolation between user applications. However, the covert channel attack does not allow accessing the data of other applications running in the same guest operating system. The same reasoning holds if the sending-end takes the form of a malicious kernel module. The affected component becomes the guest operating system, and the security authority becomes the hypervisor. The sending-end would be able to leak all the information of the guest operating system, but it would not allow accessing the data of other guests under the same hypervisor. Therefore, the fact that data is exfiltrated across virtual machines does not constitute a change of scope. The sole purpose of a covert channel attack is to exfiltrate information, or carry out modifications as instructed by the other communicating entity. Any exploit built on top of the covert channel attack (e.g. privilege escalation) is beyond the scope of this study.

3.3.5 Confidentiality Impact

Confidentiality impact assesses the severity of a disclosure of information, as well as the quantity of information that can be leaked. This criterion can be rated as "none", "low" when the attacker can only access a small amount of data and loss of this data does not result in serious consequences, or "high" otherwise.

Confidentiality impact is rated as "high". Covert channels intend to leak a selected amount of information rather than the entire set of system files. However, a successful attack against a public cloud instance can have a significant impact on a victim, such as theft of proprietary information, leakage of personal data, or theft of cryptographic keys. Microarchitectural covert channels are particularly interesting when there is no alternative means of leaking information in a non-conspicuous manner, e.g. to avoid generating network

traffic and associated logs [135]. They are relevant with advanced persistent threats, where the attacker employs cutting-edge techniques in order to maintain long-term intrusion and data exfiltration capabilities. Therefore, they are ideal candidates for stealthy leakage on high-profile targets, from attackers having other incentives than simple financial gain [85].

3.3.6 Integrity Impact

Integrity impact measures the attacker's capability to tamper with the victim's data. It can be rated as "none", "low" when the amount of data that can be modified is limited and modification of this data does not result in serious consequences, or "high" otherwise.

Integrity impact is rated as "low". The attacker can issue modifications to be applied to the victim's environment, although this requires bi-directional communication, as well as the ability to instruct data tampering operations. Such a covert channel was demonstrated by Maurice et al. [101], who managed to establish a rogue Secure Shell (SSH) connection between two AWS EC2 instances. Data modification is therefore possible, however it remains a specific case, the primarily objective being data exfiltration.

3.3.7 Exploit Code Maturity

Exploit code maturity evaluates the state of an attack, from a conceptual exploit to a fully autonomous malware. Exploitability can be rated as "unproven", "proof-of-concept" when the attack has been demonstrated but is not practical, "functional" when the exploit works in most systems where the vulnerability is present but is still not widely accessible, or "high" otherwise.

Exploit code maturity is rated as "proof-of-concept". The state-of-the-art covert channels surveyed in this paper all demonstrate a functional attack in a virtualised environment. However, researchers rarely disclose their full source-code. Therefore, current microarchitectural covert channels are not directly applicable without a skilled attacker.

3.3.8 Report Confidence

Report confidence assesses the credibility of the source which reported the vulnerability. This criterion is rated as "confirmed" when originating from a publication, "reasonable" when multiple non-official sources reported the vulnerability, or "unknown" when a single non-official source is involved.

Report confidence is rated as "confirmed". A research publication is considered an official source which is corroborated by multiple experts. As per the CVSS specification [113], disclosure of an exploit in external events such as scientific conferences automatically grants this criterion the selected rating.

3.3.9 Privilege Required

Privilege required evaluates the level of privileges that the adversary must acquire before launching the attack. This criterion can be rated as "none", "low" if privileges that allow performing basic user operations are required (e.g. changing settings), or "high" for administrative privileges. This criterion is relative to the covert channel's sending-end concealed in the victim's environment.

Exploits which rely on social engineering are rated as "none" [113]. However, the works of Ristenpart et al. [124] and Xu et al. [166] require accessing privileged page tables in order to find congruent addresses, and are thus rated as "high". All remaining covert channels can be carried out by non-privileged users.

3.3.10 Remediation Level

Remediation level accounts for potential countermeasures. This criterion can be rated as "unavailable", "workaround" for non-official mitigation, "temporary fix" for official but not permanent countermeasures, or "official fix" otherwise.

Defense techniques may rely on an ISA extension, a modification of hardware-enforced algorithms ans policies, or an entirely new hardware de-

sign. This type of countermeasure cannot be deployed as easily as a software update, and the performance cost can become too significant. For instance, Wang et al. [154] suggested a new design of the memory controller which enforces temporal isolation among different security domains. While effective, this technique results in a 150% overhead. As a result, hardware-enforced countermeasures are rated as "unavailable".

Other remediation strategies may be enforced by the cloud provider, which owns the processing, storage, network, and virtualisation components (see Table 2.1). The AWS EC2 and GCE services propose a type of instance where the user runs on a platform that is isolated from other users [134, 32]. Dedicated instances have a significant cost, e.g. an on-demand EC2 a1.2xlarge instance costs 0.204 USD per hour while a dedicated EC2 a1.2xlarge instance costs 2.2162 USD per hour. Therefore, this approach is only valid for specific, sensitive workloads. Alternatively, cloud providers have reportedly encouraged the disabling of SMT in order to prevent core-level timing channels [98]. Nevertheless, it is observed that SMT remains available on dedicated instances or for general-purpose workloads. This type of remediation strategy is rated as "temporary fix".

3.3.11 Hardware Configuration

Hardware configuration specifies the attacker's proximity with regard to the victim's VM. Covert channels can require both VMs to be scheduled on the same core, on the same processor, or on the same system. Accordingly, hardware configuration can be rated as "core", "processor", or "system". A "system" rating makes for a higher severity score, as it is easier to achieve system-level co-residency.

Ristenpart et al. [124] and Xu et al. [166] used a busy-loop mechanism to synchronise receiver and sender, implying that a CPU-level resource was shared. Therefore, these attacks cannot be considered cross-core. Similarly, the memory order buffer attack [140] requires both entities to share CPU resources. These attacks are set to "core".

Other LLC-based covert channels are set to "processor". A LLC cannot

be shared across processor dies. The memory controller attack [133] exploits a system-level component, however it was not demonstrated on a multi-processor system. These covert channels are also rated as "processor".

Finally, the DRAM row-buffer [122] attack can transmit data across processors as DRAM memory is shared at system-level. Similarly, the memory bus covert channels [164, 132] exploit the bus lock vulnerability which has system-wide effects. As such, these attacks are rated as "system".

3.3.12 Initialisation

Initialisation evaluates whether a covert channel attack requires the sender and receiver to perform an initialisation phase before leaking the victim's data. This criterion can be rated as "mandatory" or "optional". In the latter case, the covert channel remains functional in the absence of an initialisation phase, which increases the severity score. The absence of initialisation eases the deployment of the attack and decreases visible side-effects (e.g. large memory footprint), hence a greater score.

Only the memory bus [164, 132] and the memory order buffer [140] covert channels can be rated as "optional". Every other covert channel requires an initialisation phase, and are thus rated as "mandatory" for this criterion.

3.3.13 Covert Channel Capacity

A communication channel can be subject to noise. In order measure the quantity of information that can effectively be transmitted, covert channels are modelled as binary symmetric channels. Under the binary symmetric model [34], the amount of information that can be reliably transmitted is given by the channel capacity C. It is a function of the entropy binary H_2 and the raw bit rate r,

$$C = r(1 - H_2) \tag{3.1}$$

The binary entropy H_2 is a function of the bit error probability p, and is defined as follows,

$$H_2 = -p \log_2 p - (1-p) \log_2 (1-p) \tag{3.2}$$

Under the binary symmetric model, a channel behaves as follows,

- If $p = 0$, then the probability of a bit being correct is $1 - p = 1$, the binary entropy is $H_2 = 0$, and the channel capacity is $C = r$.
- If $p = 0.5$, then the probability of a bit being correct is $1 - p = 0.5$, the binary entropy is $H_2 = 1$, and the capacity is $C = 0$.

The error probability is determined by counting the number of bit flips in comparison to the original bit stream, divided by the number of bits transmitted. For instance, if there are 128 erroneous bits in a 256-bit long message, then $p = 128/256 = 0.5$. At this point and under the binary symmetric model, the channel capacity becomes null. For any $p \geq 0.5$, the covert channel is considered to be completely unreliable.

3.4 Covert Channel Scoring System

In order to provide a classification of covert channels, a new scheme must be devised, which accounts for required privileges (Section 3.3.9), remediation level (Section 3.3.10), hardware configuration (Section 3.3.11), initialisation (Section 3.3.12), and the channel capacity. These criteria are specific to the covert channel considered, and provide a point of comparison for the adversary model. The CCSS is by no means a representation of the severity of the attack. Instead, it should be taken as a complement to the CVSS which cannot solely be used to classify cross-VM covert channels.

Criteria scores have been selected such that the scoring equation is as uniform as possible. That is, the five criteria all have the same weight. The motivation behind this decision is that the importance of one factor over another is subjective. For example, one could give a higher weight to the channel capacity, arguing that communication speed and robustness is the

most important. From one perspective, this is true. A set of log, data, and application files of a password manager (∼1 GB) would take 99 days 10 hours and 5 minutes at a bitrate of 1 Kb/s to be transmitted, and 2 hours and 23 minutes at a bitrate of 1Mb/s. Cloud instances are rescheduled onto different platforms depending on resource availability and demand. Therefore, the communication speed is critical. However, from another perspective, this may be false. Faster communication rates are usually achieved by covert channels that exploit microarchitectural components closer from the execution units, which can easily be addressed by existing countermeasures (i.e. disabling SMT), or that have been extensively studied and resulted in multiple countermeasure proposals [33, 53, 78, 90, 152, 155, 158, 181]. Thus faster covert channels may not even be practical.

Furthermore, improving an evaluation scheme is usually performed over time by comparing the scores with the reality. For instance, the HeartBleed vulnerability was given a medium severity score of 5.0 in CVSS v2. Yet, it could easily be exploited and had a significant impact on industry. It now has a high severity score of 7.5 in the CVSS v3.1 To the best of our knowledge, no covert channel exploit has been reported so far. Therefore, we consider that starting with an impartial scoring equation for the CCSS is the best approach. The scoring equation is,

$$Score = 2 \times (PR + RL + HC + IN + CS) \tag{3.3}$$

Where the *PR*, *RL*, *HC*, and *IN* criteria are given a score between 0 and 1,

- *PR* (Privilege Required): 0 for "Privileged", 1 for "Unprivileged"
- *RL* (Remediation Level): 0 for "Temporary fix", 0.5 for "Workaround", 1 for "Unavailable"
- *HC* (Hardware Configuration): 0 for "Core", 0.5 for "Processor", 1 for "System"
- *IN* (Initialisation): 0 for "Mandatory", 1 for 'Optional"

And the channel capacity score CS is modelled as an affine function between the highest and lowest channel capacity observed in this study, such that it outputs a score between 0 and 1,

$$CS = \frac{1}{1.06e06 - 1.77} \times Capacity \tag{3.4}$$

The resulting CCSS score varies between 0 and 10.

3.5 Evaluation Results

Figure 3.2 represents the score of each covert channel under the CCSS and the CVSS. Results are also reported in Table 3.2. Due to missing information, Ristenpart et al.'s attack [124] was assigned an error rate of 22%, i.e. the maximum error rate observed in this study. Highest scores are achieved by the memory bus [164, 132] and DRAM row-buffer [122] covert channels. These were able to reach high-speed effective communication rates while minimising operational constraints. Meanwhile, LLC-based covert channels tend to achieve lower severity scores, due to the necessity of finding congruent addresses as well as the LLC locality.

The CVSS scores were computed with the CVSS v3.1 equations [113]. According to this study, microarchitectural covert channels achieve a medium severity score ranging from 4.2 to 5.0. It shows that covert channels in

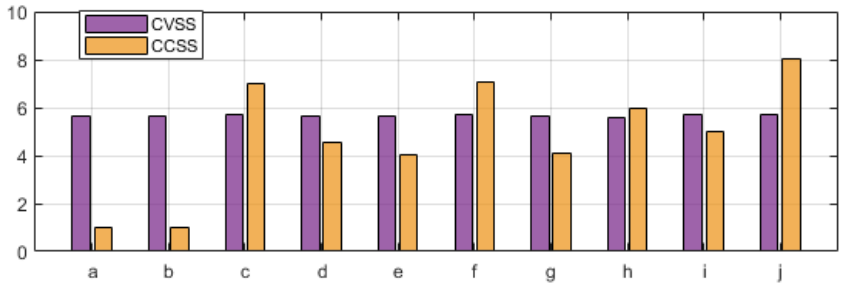


Figure 3.2: Scoring of cross-VM covert channel attacks under the CVSS and CCSS: **a** = LLC [124], **b** = LLC [166], **c** = Memory bus [164], **d** = LLC [92], **e** = LLC [100], **f** = Row-buffer [122], **g** = LLC [101], **h** = Memory order buffer [140], **i** = Memory controller [133], **j** = Memory bus [132]

Table 3.2: Scoring of cross-VM covert channel attacks under the CVSS and CCSS.

Attack	Exploited resource	Error	Capacity	CVSS	CCSS
[124]	Last-level cache	-	0.2 bps	4.2/10	1.6/10
[166]	Last-level cache	9.28%	1.77 bps	4.2/10	1.6/10
[164]	Memory bus	0.39%	330 bps	5.0/10	6.7/10
[92]	Last-level cache	22%	287 kbps	4.9/10	4.3/10
[100]	Last-level cache	5.7%	514 bps	4.9/10	3.7/10
[122]	DRAM row-buffer	0.4%	573 kbps	5.0/10	6.8/10
[101]	Last-level cache	0%	45.2 kbps	4.9/10	3.8/10
[140]	Memory order buffer	∼5%	1.06 Mbps	4.8/10	5.7/10
[133]	Memory controller	7.8%	90.7 bps	4.9/10	4.7/10
[132]	Memory bus	5.4%	333 bps	5.0/10	8.0/10

IaaS are practical, that they should not be overlooked, and that suitable countermeasures should be devised in the short term in order to tackle timing channel vulnerabilities. More specifically, addressing the DRAM row-buffer and memory bus covert channels should be prioritised, as cache-based covert and side channel attacks have already been extensively studied.

When comparing the two evaluation frameworks, the CCSS outlines disparities among covert channel attacks which the CVSS does not. This is demonstrated through the difference in variance among the different scores. Table 3.3 shows that the CVSS and CCSS achieve respectively 4.77 and 4.69 score averages, yet the variance is significantly more important for the CCSS (4.60 against 0.09 for the CVSS). The variance is a measure of dispersion, representing how far samples are spread out from the average. The CCSS effectively shows that all microarchitectural covert channels do not follow the same adversary model and have heterogeneous operational constraints.

Table 3.3: Average (μ) and variance (σ^2).

	CVSS score	CCSS score
μ	4.77	4.69
σ^2	0.09	4.60

For example, the works proposed by Ristenpart el al. [124] and Wu et al. [164] would both be rated as medium severity vulnerabilities under the CVSS. Yet, the former attack has significant shortcomings including obtaining privileges, achieving core-level co-location, finding congruent addresses, and a low communication speed. Thus the resulting CVSS scoring of the covert channel proposed by Ristenpart et al. [124] as a medium severity vulnerability is not adequate. In comparison, the proposed evaluation framework successfully highlights the benefit of one covert channel over another, with respective scores of 1.6 and 6.7. This shows that the evaluation of microarchitectural covert channels cannot be performed entirely based on the current industry standard, and that the criteria studied in the CCSS should be accounted for when devising new cross-VM covert channel attacks.

3.6 Summary

In this chapter, state-of-the-art cross-VM microarchitectural covert channels were evaluated against the CVSS3.1 scoring framework, revealing medium severity scores ranging from 4.2 to 5.0. In comparison, the SSLv3 POODLE [110] and OpenSSL Heartbleed [109] vulnerabilities respectively achieved scores of 3.4 and 7.5. These were patched shortly after their disclosure. Services built on cloud computing continue offering guarantees on the confidentiality and integrity of their customers' data and there are still no practical countermeasures against microarchitectural covert channel attacks released several years ago [122, 164]. The loss of data, e.g. under General Data Protection Regulation (GDPR) requirement, could result in dramatic consequences for the cloud provider, the software provider, and their customers.

A new evaluation framework was then developed, named Covert Channel Scoring System, with the intent to devise metrics for fair comparison and uncover potential research gaps. It is the first scoring system that accounts for both operational constraints and performance (speed and robustness), thus making it a highly relevant framework for assessing the severity of leakage channels in cloud environments. The analysis revealed that the fastest covert channels are not necessarily the most eminent attacks, as they usually as-

sume a close locality between sender and receiver, or a complex initialisation phase, resulting in lower severity scores. Achieving higher communication rate should not come at the cost of unreasonable operational constraints. This sets a direction for further work, where considerations for the adversary model are re-aligned with the reality of the commercial environment.

Chapter 4

The Memory Controller-based Covert Channel

Contents

In this chapter, a new covert channel is devised based on the modulation of the integrated memory controller. This attack allows establishing a rogue communication channel across processes running on separate processor cores and across VMs. The covert channel is compared to related literature, and mitigation strategies are discussed.

4.1 Introduction

The functional behaviour of a system is usually well understood by designers. For instance, the seL4 micro-kernel has been proposed as a general purpose solution, providing strong assurance of confidentiality, availability, and integrity enforcement from a functional perspective [80]. The identification of hidden leakage channels works by analysing the system's resources or source-code. Yet, these identification methods rarely account for the system's temporal behaviour. Murray et al. [105] highlighted that seL4 micro-kernel formal proofs completely omit timing channels. The gap left by the absence of temporal behaviour characterisation allows devising new timing channels, eventually allowing an attacker to breach information flow policies that enable secure computation on multitenant platforms.

Percival [121] demonstrated a covert channel between two threads, based on contention within the L1-D and L2 caches. Shortly after, Wang and Lee [157] designed a covert channel that leverages contention on multipliers. More recently, Sullivan et al. [140] demonstrated a high-speed covert channel between two hyperthreads in AWS EC2 and Google Compute Engine instances. In a virtualised environment, core-level co-residency is hard to achieve as VMs tend to be isolated onto different cores. Furthermore, this class of covert channels is only relevant to cloud platforms where SMT is enabled. In reality, this condition rarely occurs. Cloud providers commonly disable support for SMT [98] as well as memory deduplication [17], thus hindering a large range of microarchitectural timing attacks.

Other works proposed exploiting the LLC cache as it is shared among cores. Xu et al. [166] proposed exploiting conflicts in the LLC. They used a covert channel to achieve co-location in an Amazon EC2 setting. Further works followed based on the Prime+Probe [92, 100, 115] or Flush+Flush technique [56]. Maurice et al. [101] implemented a robust covert channel capable of establishing a rogue SSH connection across AWS EC2 instances. A number of academic works have been proposed in order to tackle timing vulnerabilities emanating from the cache, including hardware cache partitioning [90, 118, 121, 158], software cache partitioning [33, 53, 78], and noise

injection [152, 158]. It is difficult to assess whether these covert channels could bypass such countermeasures, and calls for further analysis. Also, Section 3.5 showed that exploiting cache memories requires finding congruent addresses between the two communicating entities, and the locality of the cache can severely restrict the feasibility of the attack.

Other internal (memory controller, on-chip memory bus) and external (DRAM) resource remain potentially exploitable [108, 122, 154, 156, 176]. The vulnerability of the memory controller was also demonstrated by Moscibroda et al. [108]. Their work shows that by combining all timing channels detailed in Section 4.2, a malicious process can slowdown the execution of a concurrent process by a factor of 190%. It is worth noting that their DoS attack exploits both the memory controller, and the DRAM row-buffer. Furthermore, they do not address the problem of encoding and decoding information across virtual machines via the channel scheduler. Pessl et al. [122] built a high-speed covert channel based on on the DRAM row-buffer. Their channel reaches up to 596 kbps in virtualised environment. Mitigating row-buffer covert channels could be achieved by enforcing a close-page policy on the memory controller. As a result, every memory access would result in a row-miss, thus inhibiting the timing channel. Furthermore, authors relied on a privileged adversary model, and both entities need to undergo an initialisation phase in order to agree on a specific DRAM bank. This agreement cannot be performed online without incurring additional memory usage side-effects. In this chapter, the very first instance of a memory controller-based microarchitectural covert channel is presented. Eventually, the attack allows an adversary leaking information across processor cores and VMs, thus bypassing information flow policies essential to the security of multi-tenant computing platforms. The proposed covert channel is tested on three Intel x86 microarchitectures, namely Ivy Bridge, Broadwell, and Skylake. The work presented in this chapter is based on publication [133].

4.1.1 Structure of this Chapter

Section 4.2 provides additional background on the memory controller, and analyses the different sources of contention.

Section 4.3 presents the first instance of the proposed attack. It consists of a cross-core covert channel established between two native processes running with privileges. The threat model, attack principle, and implementation considerations are examined.

Section 4.4 presents the second instance of the proposed attack. It consists of a cross-core covert channel established between two virtualised processes running without privileges. This instance opens the door to cross-VM attacks in cloud environments. As in the previous section, the threat model, attack principle, and new implementation considerations are described.

Section 4.5 describes the experimental setup, and evaluates the capacity of both covert channels under the binary symmetric model.

Section 4.6 discusses potential countermeasures. Finally, Section 4.7 summarises the contributions of this chapter.

4.2 Sources of Contention

The working principle of memory controllers is introduced in Section 2.2.5. Delays can be generated via the bank scheduler (see Figure 4.1), as requests from different processes are mixed in the same bank buffer. If process A is the only one requesting data in a bank, its memory accesses will be served immediately. However, if another process B starts requesting data in the same bank as process A, requests of A and B will compete for scheduling. Because the load on the bank scheduler increases, requests of process A can be delayed.

Delays may also be introduced via the channel scheduler, since it arbitrates requests for several banks. If there are no other requests than for bank i, these will systematically win arbitration and be served immediately. However, if other requests for bank j, with $j \neq i$, compete for access to the channel, the load on the channel scheduler will increase, resulting in requests

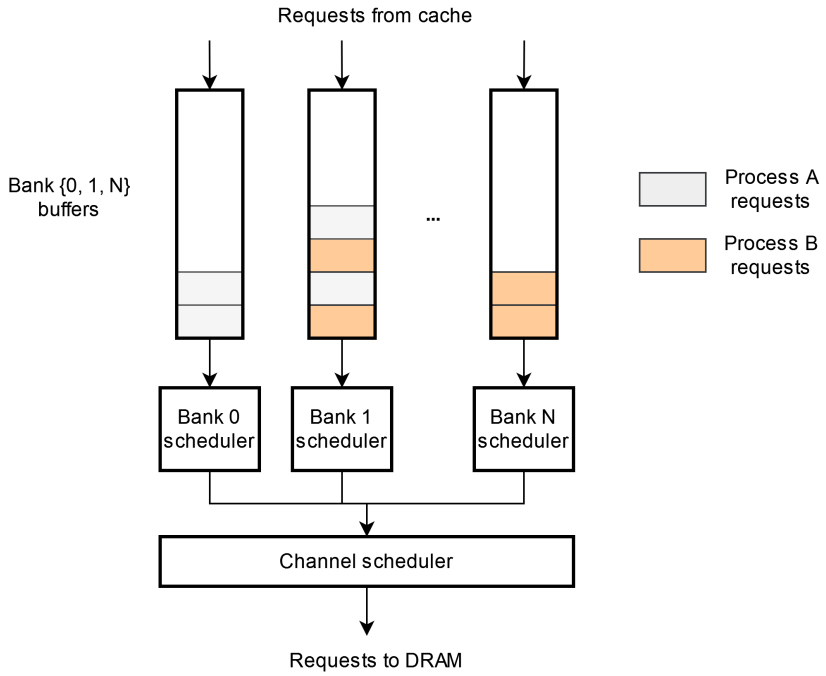


Figure 4.1: Memory controller representation.

for bank i to be delayed.

Finally, contention can be introduced via the DRAM row-buffer. Within the same bank, if there are no other requests than for row m, these will systematically result in row-hits. However, if other requests in row n interfere, with $m \neq n$, the row-buffer will alternatively be updated with rows m and n, resulting in frequent row-misses.

4.3 Privileged Native Covert Channel

This section presents the basic concept to generating contention via the channel scheduler through a privileged covert channel in a native environment.

4.3.1 Threat Model

The threat model assumes two processes, a *receiver* and a *sender*, willing to share information. The security policy forbids these two entities from com-

municating directly. The sender possesses sensitive information and intends to transmit this information to the receiver. Entities each have their own address space, which is dissociated in physical memory. They are running on different cores. Both entities require root privileges, and have knowledge of the DRAM addressing functions.

4.3.2 Principle

The proposed covert channel exploits timing variations upon uncached memory accesses. The receiver and sender both occupy space in X, the set of DRAM banks served by a single channel. The receiver "listens" to the channel by continuously performing uncached memory accesses at a predetermined address, i.e. bank x_0 with $x_0 \in X$. The sender writes on the channel by creating conflicts on the resources involved in the memory accesses of the receiver. The sender generates bit values as follows,

- A zero is written by performing uncached memory accesses in bank x_1, with $x_1 \neq x_0$ and $x_1 \in X$. Because the channel scheduler only serves banks x_0 and x_1, contention is negligible. Thus, the receiver's memory access in bank x_0 will result in a "normal" latency, which is interpreted as a zero.

- A one is written by performing uncached memory accesses in all banks x comprised in X, except for bank x_0. This operation causes the channel scheduler to serve requests for every bank within X, which generates an observable contention. Thus, the receiver's memory access in bank x_0 will increase in latency, which is interpreted as a one.

Algorithm 1 summarises how bit values are encoded and decoded across the native covert channel. An Access(x) operation consists in performing an uncached memory access in bank x. A Probe(x) operation is equivalent, at the exception that the elapsed time of the operation is returned to its caller. In order to write a zero, the sender needs to perform memory accesses in a different bank than the receiver. Doing so prevents interference from the DRAM row-buffer. Indeed, if both entities were to read from the same bank,

Algorithm 1: Memory Controller Native Covert Channel Protocol

x: DRAM bank;
X: set of DRAM banks x;
N: number of bits to send;
$send[N], recv[N]$: respective buffers of sender and receiver;

Sender	**Receiver**
for all $i \in [0; N]$ **do**	**for all** $i \in [0; N]$ **do**
if $send[i] == 1$ **then**	{Timed access to one bank}
{Access many banks at once}	$t =$ Probe(x_0);
Access($x_1, ..., x_{X-1} \mid x \neq x_0$);	**if** $t > threshold$ **then**
else	$recv[i] = 1$;
{Access one bank}	**else**
Access($x_1 \mid x_1 \neq x_0$);	$recv[i] = 0$;
end if	**end if**
end for	**end for**

they would most likely read from different rows (a bank contains thousands of rows). As a result, reading alternatively from the sender's row and the receiver's row would cause the row-buffer to be systematically updated. Thus, memory accesses would result in a majority of row-misses, and dramatically increase in latency. Because this attack exploits exclusively the memory controller, such interference is undesirable. Furthermore, it is preferable to keep the sender active upon sending a zero, in order to compensate the effect of other microarchitectural components (e.g. memory bus). The objective is to demonstrate the vulnerability in the memory controller, therefore its effect must be isolated from other sources of contention.

4.3.3 Design Considerations

The mechanism for exposing timing variations is relative to the microarchitecture, not the above OS and applications. Therefore, porting this implementation from Linux to Windows or MacOS would not benefit the study in any way. The native, privileged covert channel works in two phases. First, each entity must identify a virtual address which maps to the desired DRAM bank(s). Then, both processes synchronize to exchange information covertly.

In the first phase, processes read the restricted */proc/self/pagemap* page

translation table to compute the pointer's physical address. Physical-to-DRAM address translation (channel, rank, bank, row, column) requires knowledge of the DRAM addressing functions. These vary from one processor to another, and must be reverse-engineered if not disclosed by the manufacturer[1]. The DRAM address mapping was computed with the reverse engineering tool first presented in [122]. Prior to launching the covert channel, entities can decide on which DRAM banks to use specifically.

In the second phase, entities use the operating system wall-clock to synchronize. The `clflush` instruction is used to flush the cache upon each memory access, so as to force the request to be served from DRAM. Because an uncached memory access is higher in latency than a cached one, the `cpuid` instruction is used to prevent out-of-order execution of the time-stamp reads. Finally, time-stamps are read with the `rdtsc` and `rdtscp` instructions[2].

4.4 Unprivileged Cross-VM Covert Channel

This section presents the application of the memory controller covert channel to an unprivileged attacker in a cross-VM scenario.

4.4.1 Threat Model

Similarly to the threat model presented in Section 4.3.1, the sender and receiver are willing to share information illegitimately. However, they are now running in separate VMs, with each VM having a dedicated address space. The hardware platform features a multi-core processor, such that the hypervisor schedules each VM on a different core. The security policy enforced ensures memory isolation, access control, and does not present any software vulnerability. The sender and receiver are both unprivileged user programs. Memory accesses are handled by the guest operating system, itself managed by the hypervisor or host operating system.

[1]DRAM addressing functions on the Ivy Bridge test platform (see Table 4.1): BA0= $b_{13} \oplus b_{17}$; BA1= $b_{14} \oplus b_{18}$; BA2= $b_{16} \oplus b_{20}$; and Rank= $b_{15} \oplus b_{19}$

[2]The source code of the native covert channel is available at `https://github.com/bsepage/mc2c.git`

4.4.2 Principle

In Section 4.3, the covert channel is designed for a privileged adversary model. Indeed, an unprivileged attacker is unable to read the */proc/self/pagemap* file, which is necessary for virtual-to-physical address resolution. Yet, the attacker needs to find addresses in its virtual address space which map to different DRAM banks.

Rather than searching for specific banks in a process' address space, several virtual pages were mapped in order to resolve how these are spread across physical memory. For each virtual page, the base pointer was translated into a physical address, and then converted into bank address, according to the platform's DRAM addressing functions. The following observations were made,

1. A single (page-aligned) virtual page is mapped to a single bank.
2. Virtual pages within the same logical address space are mapped to different DRAM banks.

These observations suggest that the sender only requires allocating multiple virtual pages, and that each page will map to a different bank. However, there is a probability that one page will be mapped to the same bank as the one accessed by the receiving-end. Such scenario would cause row-buffer conflicts to occur. Accessing different rows triggers row-buffer updates, which would add a significant delay into the receiver's accesses. Algorithm 2 summarises how bit values are encoded and decoded across the cross-VM covert channel. The receiver "listens" to the channel by continuously performing uncached memory accesses at a fixed random location, i.e. virtual page r_0. Given a set S of virtual pages s in the sender's virtual address space, the sender generates bit values as follows,

- A zero is written by performing uncached memory accesses in virtual page s_0, with $s_0 \in S$. Because the channel scheduler serves at most two banks (i.e. one mapping to s_0 and one mapping to r_0), contention is negligible. Thus, the receiver's memory access in bank r_0 will result in a "normal" latency, which is interpreted as a zero.

Algorithm 2: Memory Controller Virtual Covert Channel Protocol

r: virtual page in receiver's address space;
s: virtual page in sender's address space;
S: set of virtual pages s;
N: number of bits to send;
$send[N], recv[N]$: respective buffers of sender and receiver;

Sender	**Receiver**
for all $i \in [0; N]$ **do**	**for all** $i \in [0; N]$ **do**
if $send[i] == 1$ **then**	{Timed access to one page}
{Access many pages at once}	t = Probe(r_0);
Access($s_0, ..., s_{S-1}$);	**if** $t > threshold$ **then**
else	$recv[i] = 1$;
{Access one page}	**else**
Access(s_0);	$recv[i] = 0$;
end if	**end if**
end for	**end for**

- A one is written by performing uncached memory accesses in all banks s comprised in S. This operation causes the channel scheduler to serve requests in many banks at once, which generates an observable contention. Thus, the receiver's memory access in bank r_0 will increase in latency, which is interpreted as a one.

Figure 4.2 shows the latency of the receiver's memory accesses, with the sender alternatively being active and inactive. The latency graph shows that when the sender is active, the receiver presents an overhead of 6.5 CPU cycles on its accesses. The timing variation indicates that the proposed strategy is valid for creating a covert channel. This new approach has the benefit that it completely discards the virtual-to-bank address translation procedure. Therefore, the attacker neither requires privileges, nor knowledge of the platform's DRAM addressing functions. In this configuration, the attack can be applied to virtual environments, where physical addresses are virtualized by the hypervisor.

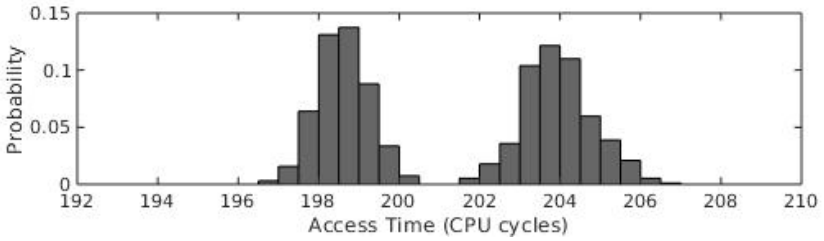


Figure 4.2: Effect of active sender upon latency of receiver's memory accesses (Ivy Bridge setup).

4.4.3 Design Considerations

The cross-VM, unprivileged covert channel works in two phases. In the first phase, the sender maps and locks memory pages without reverse-engineering their physical location. Note that the sender and receiver no longer require agreeing on specific DRAM banks. In the second phase, entities respectively read or probe their memory accesses to encode and decode bit values. Probing and accessing is performed using the `clflush`, `cpuid`, `rdtsc`, and `rdtscp` instructions.

4.5 Evaluation

This section presents the evaluation of the covert channel capacity under the binary symmetric model, in both native and virtualised environment. Three Intel x86 microarchitectures are tested.

Table 4.1: Experimental setups.

Setup	Processor	CPU Frequency	Memory	#DRAM banks
Ivy Bridge	Intel i5-3210M	2.5 GHz	1×4GB DDR3	16
Broadwell	Intel i7-5500U	2.4 GHz	1×8GB DDR3	16
Skylake	Intel i5-6300U	2.4 GHz	1×8GB DDR4	16

4.5.1 Experimental Setup

The experimental setups used for characterising the channel capacity is presented in Table 4.1. The Kernel Virtual Machine [96] hypervisor is used to manage virtual machines, and each VM is operated by a Debian distribution (Linux kernel version 4.19.0). All setups feature a dual-core processor, allowing us to lock the sender's and the receiver's VM onto different processor cores. The `virsh edit` command is used to assign a specific `cpuset` to the `vcpu` attribute. All the setups feature 16 DRAM banks. Note that a commercial infrastructure- or platform-as-a-service server system will likely feature greater amounts of DRAM, i.e. the occurrence of row-buffer interference will drop accordingly. Therefore, the proposed setup represents a worst-case scenario for the attacker.

4.5.2 Channel Capacity

The channel capacity is measured by modelling the covert channel as a binary symmetric channel (see Section 3.3.13). It is a function of the raw bit rate r and the binary entropy H_2, and was calculated using Equations 3.1 and 3.2. Figure 4.3 compares the capacity C and error probability p against a raw bit rate r ranging from 100 bps to 1300 bps for the native scenario (Figure 4.3(a)), and from 50 bps to 350 bps for the virtualised scenarios (Figures 4.3(b), 4.3(c), and 4.3(d)). Measures were taken by sending a fixed-size message and counting the number of bit flips on the receiving-end. The error probability p was then calculated as the number of bit flips divided by the number of bits sent.

In the native scenario (Figure 4.3(a)), the error probability stays below 0.1 for bit rates of up to 1250 bps. The channel capacity reaches up to 729 bps, with an error probability of 6.25%. In the virtualized scenarios, the Ivy Bridge (Figure 4.3(b)), Broadwell (Figure 4.3(c)), and Skylake (Figure 4.3(d)) setups respectively achieve a maximum capacity of 90, 95, and 69 bps. The error probability remains below 0.1 for a raw bit rate of up to 175 bps across the three setups. Results are reported in Table 4.2.

Virtualization has a significant impact on the effective channel capacity,

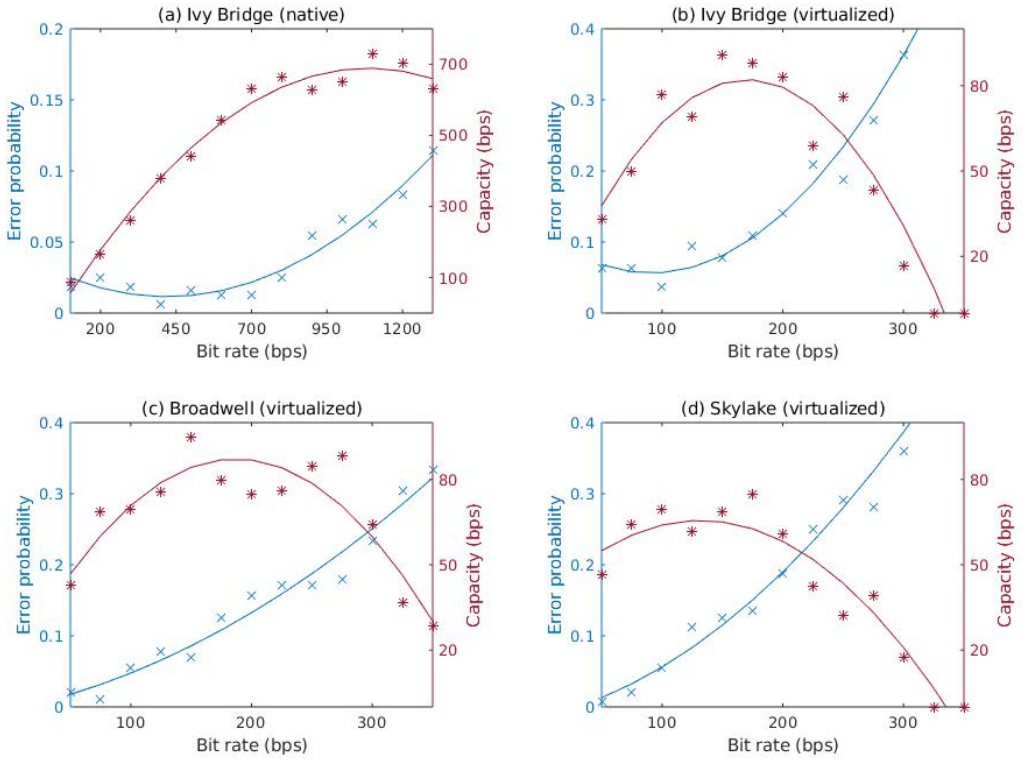


Figure 4.3: Effective capacity and error probability measured against raw bit rate.

as it brings additional sources of noise. First, sender and receiver compete with each other to be scheduled by the hypervisor. Second, the sender and receiver are not able to use the operating system wall-clock to synchronise, as they run in separate VMs. The receiver might sample at a different rate than the sender can transmit, with the bias increasing over time. Third, programs executing concurrently (e.g. hypervisor) can alter the state of the channel scheduler, bank scheduler, or row-buffer. Therefore, it is normal that the cross-VM version of the covert channel achieves lower performance than its native counterpart.

Table 4.2: Experimental results.

Setup	Environment	Bit rate	Error rate	Capacity
Ivy Bridge	Native	1100 bps	6.25%	729 bps
Ivy Bridge	Virtualized	150 bps	7.8%	90 bps
Broadwell	Virtualized	150 bps	7%	95 bps
Skylake	Virtualized	100 bps	5.6%	69 bps

4.6 Mitigation

Section 3.5 showed that the memory controller covert channel achieves a severity score of 4.7 (CCSS), making it more practical than any known cache-based covert channels. Potential mitigation strategies are discussed here.

Memory controller-based (and DRAM row-buffer) covert channels rely on uncached memory accesses. Therefore, one countermeasure consists in disabling or restricting access to the `clflush` instruction. This mitigation technique would require architectural changes resulting in increased complexity. In addition, it is still possible to invalidate cache lines by priming a cache set or a cache line, as in the Prime+Probe and Evict+Time attacks.

Auditing-based techniques have been proposed in the past [50]. The systematic flushing of the cache causes a very high number of cache misses, which can be monitored in order to detect abnormal behaviours. However, auditing usually results in high numbers of false positives. Further work is required to assess whether this is a suitable approach.

Wang et al. [154] proposed an alternative hardware design of a memory controller. They achieve temporal isolation between different security domains, at the cost of a memory latency ranging from 60% to 150%. So far, there hasn't been any countermeasures relying on spatial isolation.

4.7 Summary

This chapter presented two instances of microarchitectural covert channel attacks using the integrated memory controller. The first attack is privileged and was tested in a native environment. It achieved a capacity of up to 729

bps (raw bit rate of 1100 bps). The second attack is unprivileged and was tested in a virtualised environment. It achieved a capacity of up to 95 bps (raw bit rate of 150 bps). The measurement study presented in Chapter 3 showed that the memory controller covert channel obtains a CCSS score of 4.7, where cache-based covert channels never go as high as 3.8. CVSS scoring showed similar severity impacts, with 4.9 for the proposed attack against 4.9 and 4.7 for cache-based counterparts (see Table 3.2). The proposed covert channel is a trade-off between performance and considerations for operational constraints. The results of this work have been reported to Intel for responsible disclosure.

Future work should address the problem of finding countermeasures that prevent exploitation of the memory controller and the DRAM row-buffer resource. In parallel, this attack should be further developed in order to expand it to multi-processor platforms, where the communicating entities belong to different NUMA nodes. Mechanisms for bi-directional communication should also be investigated, thus allowing the implementation of more advanced protocols. As mentioned earlier, the vulnerability exposed in this chapter is independent from the operating system. However, the generalisation of this attack to other ISA such as ARM or RISC is required, as the memory controller implementation will have different properties. The memory controller may present other vulnerabilities on ARM- and RISC-based microarchitectures.

Chapter 5

Eluding Defences with a Memory Bus-based Covert Channel

Contents

In this chapter, a new instance of the memory bus covert channel is proposed, which challenges the set of countermeasures against timing channels, whether these are already deployed or still theoretical. The resulting implementation shows that x86 microarchitectures still present salient vulnerabilities, and that state-of-the-art defence strategies—even theoretical ones—remain unsuccessful at hindering data leakage in multi-tenant environments.

5.1 Introduction

Microarchitectural covert and side channels have been increasingly popular in the last decade, and even more since the release of the Spectre and Meltdown attacks [81, 89]. In response, a plethora of mitigation strategies has been proposed in academia, from new hardware designs through software partitioning to anomaly detection. These defence strategies often aim at closing a Prime+Probe [116] covert or side channel, omitting attacks which are not based on cache exploitation. In parallel, new attacks consider a set of countermeasures, usually the ones already deployed in the targeted environment, and aim at demonstrating a residual threat despite these existing countermeasures. A trend that is observed is that attackers often disregard the latest developments in terms of defences, arguing that these are not deployed by OS or cloud providers. Therefore, it is difficult to assess the novelty of these attacks, as they might already have been addressed by recent works.

In this chapter, a retrospective analysis on state-of-the-art attack and defence techniques is performed. To do so, a new microarchitectural covert channel is proposed which allows cross-VM communication in a public cloud, while discarding the usage of artifacts which are theoretically made unavailable by recently proposed countermeasures. Covert and side channel attacks differ in the attack scenario, however they share the underlying mechanisms for leaking information. Therefore, the study accounts for all defence strategies, as long as they are relevant with the covert channel attack scenario (e.g. constant-time techniques).

Auditing strategies have been proposed against timing channels [175, 9, 31, 84]. These aim at detecting abnormal behaviours at runtime, and deploying reactive measures accordingly (e.g. interrupting the suspected workload, migrating a VM, temporarily injecting noise, etc.). Because the sustainability of the auditing approach is highly correlated to the ability of avoiding false positives, multiple machine learning-based techniques have also emerged [119, 8, 106, 23, 107]. The main drawback of auditing is that it is usually tailored for specific workloads such as cryptographic computations. Thus its applicability against microarchitectural covert channels remains an open

question, as they might not have an easily identifiable signature. Furthermore, auditing does not aim at closing a malicious behaviour, but at detecting it. While the decision is made to apply reactive measures, sensitive information such as cryptographic keys might have already been leaked. Auditing strategies are not capable of closing a microarchitectural covert channel in a deterministic way, and their practicality has already been questioned due to their performance cost [56]. Therefore, they are not considered viable countermeasures against such attacks. The work presented in this chapter is based on the publication [132].

5.1.1 Structure of this Chapter

Section 5.2 presents multiple strategies to closing timing channels, from noise injection to resource partitioning. Requirements on how to bypass these defence mechanisms are extracted, in order to establish a strategy for the implementation of the proposed covert channel.

Section 5.3 presents the new instance of the memory bus-based covert channel. It discusses the adversary model and puts forward several techniques in order to render the countermeasures presented in Section 5.2 obsolete.

Section 5.4 addresses the evaluation of the proposed attack. The covert channel is modelled under the binary symmetric model and its performance are measured in a commercial environment. The effects on microarchitectural states are also discussed.

Section 5.5 suggests different approaches to closing the proposed covert channel, and performs a comparison with the ARMv8.2-A instructions in order to identify whether this instance could also be deployed on mobile platforms. Finally, Section 5.6 summarises the contributions of this chapter.

5.2 Design Goals

This section reviews the countermeasures introduced in Section 2.4.2 in order to derive design goals for the new covert channel, and performs a comparison with state-of-the art covert channel attacks.

5.2.1 Deriving Design Requirements

As surveyed in Section 2.4.2, several proposals consist of jittering the timestamps of high-resolution timers [62, 152, 99, 93], such that it prevents the receiver to obtain accurate timestamps. The first objective is thus to eliminate the usage of the `rdtsc` and `rdtscp` high-resolution timers which could become unreliable due to noise injection countermeasures. The same requirement applies to operating systems clock sources which lack accuracy. In order to account for noise injection on high-resolution timers, the following condition is set:

> **Design requirement 1: Noise injection on timers**
> The covert channel shall not rely on the `rdtsc` nor `rdtscp` instruction for measuring timing variations.

Noise can also be injected in the cache hierarchy in order to prevent the victim's accesses to conflict with the attacker's cache sets, or to prevent the attacker from distinguishing cache-hits and cache-misses [123, 44]. Eventually, this approach inhibits the attacker's capability from learning about the victim's working cache set, and hinders covert channels that rely on congruency, e.g. Prime+Probe or Evict+Reload [116]. Other works suggested "fuzzing" the cache replacement policy such that the attacker cannot have assurance of the presence of the victim's data in cache memory [91, 45, 168]. In order to build a covert channel that remains resilient in the presence of noise injection countermeasures, the second design goal consists in making the covert channel independent from the state of cache memory:

> **Design requirement 2: Noise injection on caches**
> The attacker cannot rely on the latency of cache accesses. Therefore, data caches such as the L1-D, the L2, and the LLC shall not be used as a communication medium.

The third category of countermeasure is software partitioning. As surveyed in Section 2.4.2, software partitioning includes cache colouring techniques which aim at enforcing security domains within the data caches, by

means of address resolution bits (e.g. two colouring bits allow defining four security domains) [158, 155, 78]. This approach prevents an attacker exploiting congruency. The second design choice already allows circumventing these countermeasures. However, other software partitioning techniques have been proposed, namely disabling page sharing [98] and disabling SMT [17]. The former allows hindering attacks that leverage shared memory such as Flush+Reload and Flush+Flush [56]. The latter prevents two hardware threads from exploiting contention among CPU-level resources. Therefore, the third design choice is defined as follows:

Design requirement 3: Software partitioning
The covert channel must remain functional when shared memory and SMT are disabled. Also, set-associative caches shall not be used as a communication medium.

Finally, hardware partitioning countermeasures have been studied, to provide physical isolation among multiple cache sets [37, 118, 158], to enforce time-division multiplexing in the memory controller [154], or to apply priorities to access the on-chip network [156]. Similarly to cache colouring, these techniques aim at enabling several security domains to co-exist while protecting microarchitectural resource from being abused. Gruss et al. [56] also suggested making the `rdtsc` and `clflush` instructions privileged. This consideration is also discussed in Section 4.6. In order to account for hardware partitioning, the following condition is set:

Design requirement 4: Hardware partitioning
The covert channel shall not rely on either the memory controller or the interconnect as a communication medium. Furthermore, the attacker cannot execute privileged code. The `rdtsc` and `clflush` instructions are considered privileged and are thus unavailable.

5.2.2 Comparison with State-of-the-Art

None of the LLC-based cross-VM covert channels [124, 166, 92, 100, 101] meet the second design requirement. These cannot meet requirement 3 either as they exploit caches' set-associativity. Ristenpart et al. [124] and Xu et al. [166] require accessing (privileged) page tables in order to find congruent addresses, thus they also fail to meet requirement 4.

The MOB covert channel [140] depends on the availability of SMT. This attack fails to meet requirements 1 and 3. The DRAM row-buffer [122] and memory controller [133] attacks fail to meet requirements 1 and 4. Both rely on cache flushing in order to force memory accesses being served from DRAM, and the memory controller covert channel exploits a microarchitectural component which has been discarded.

All existing covert channels rely on the `rdtsc` instruction and thus fail to meet requirement 1 and 4. Also, the memory bus covert channel proposed by Wu et al. [164] doesn't meet requirement 4 as they did not specify how they implemented uncached memory accesses—hence we assume that they proceeded with the `clflush` instruction. Liu et al. [93] claimed that their own memory bus covert channel can be closed by injecting noise in timers. We show in this paper that it is still possible to design the covert channel

Table 5.1: Cross-VM covert channel attacks against desired design goals.

Attack	Exploited resource	D1	D2	D3	D4
[124]	Last-level cache	-	-	-	-
[166]	Last-level cache	-	-	-	-
[164]	Memory bus	-	●	●	-
[93]	Memory bus	-	●	●	-
[122]	Row-buffer	-	●	●	-
[92]	Last-level cache	-	-	-	-
[101]	Last-level cache	-	-	-	-
[140]	MOB	-	●	-	-
[133]	Memory controller	-	●	●	-
[100]	Last-level cache	-	-	-	-
Proposed attack	Memory bus	●	●	●	●

while bypassing their defence strategy.

More generally, the memory bus covert channel was initially designed to overcome the drawbacks of cache-based attacks, namely the addressing uncertainty (e.g. unprivileged virtual-to-physical address translation in virtualised environment), the scheduling uncertainty (e.g. synchronisation errors), and cache physical limitations (e.g. exploiting the L1-D doesn't allow cross-core communication). In this paper, we demonstrate that it also enables bypassing state-of-the-art countermeasures against timing channels.

5.3 Building a Stealthy Covert Channel

This section presents an instance of the memory bus covert channel that meets all the requirements established in Section 5.2.

5.3.1 The Memory Bus-based Covert Channel

In a multi-threaded application, shared memory regions may be accessed concurrently. In order to prevent undesirable situations such as race conditions, instructions can be performed atomically. In an atomic memory operation, the requested cache line is locked in order to prevent its modification by another thread. A singularity occurs when accessing a memory region which spans across two cache lines.

Wu et al. [164] observed that, upon accessing a cache line-crossing region (a.k.a. exotic), atomicity was enforced by locking the memory bus. By guaranteeing exclusive access of the shared bus to one thread, others would be unable to modify the cache lines of interest. When the exotic operation is completed, the memory bus is unlocked.

Moreover, Wu et al. [164] noticed that a similar behaviour happens on NUMA architectures. Atomic accesses to exotic regions result in every outstanding load/store operation to be completed across all CPUs before the atomic operation is performed [71]. This strategy effectively guarantees that no other memory operation can affect the cache lines of interest. However, it also introduces significant timing variations which are visible across all

CPUs.

A covert channel can be created based on the effect of exotic memory accesses: a one is transmitted by generating atomic operations on a cache line-crossing region, a zero is transmitted by remaining idle for a fixed amount of time. Concurrently, the receiving-end probes its own accesses and interprets low and high latency accesses as zeroes and ones.

5.3.2 Threat Model

There are two communicating entities, a *sender* and a *receiver*. The sender exists in the victim's environment in the form of a trojan or any other form of malicious program. The receiver exists in the attacker's environment. Both communicating entities execute without privileges. The instances of the victim and the attacker are scheduled on separate cores of the same processor. The hypervisor is assumed to be free of any software vulnerability, and instances are logically separated. Thus sender and receiver do not share any memory region. Finally, it is assumed that state-of-the-art countermeasures are operating in the environment of both the sender and the receiver, and that these countermeasures impose the requirements listed in Section 5.2.

5.3.3 Implementation

Wu et al. [164] designed a cross-VM covert channel based on the memory bus lock behaviour. However, as described in Section 5.2, their covert channel can be closed with various countermeasures. Here, we demonstrate how to design the memory bus covert channel in a way that meets requirements 1 to 4. The covert channel can be broken down into three primitives:

Sending-end

In order to force atomicity, a `lock` prefix can be attached to an instruction. The lock signal can only be applied to read-modify-write operations whose destination operand is a memory location. Read-modify-write operations combine a load, an arithmetic, and a store operation. We choose the `xchg`

instruction which simply swaps the contents of its two operands, and automatically asserts a lock signal if the first operand is a memory location. In order to transmit a one, contention is generated by passing to the assembly function a pointer with a base address aligned on a cache line boundary added with an offset of 63 bytes (Listing 1). In order to transmit a zero, the same assembly function can be passed a pointer with a base address aligned on a cache line boundary. Also, promoting the operation to 64-bit wide with the `rex.w` prefix allows reducing the global time of execution by half. The full source-code of the sending-end is provided in Appendix A.

```
1  ; RDI = pointer to exotic or "normal" region
2  REX.W XCHG [RDI], RAX   ; read-modify-write operation
3  RET
```

Listing 1: Transmitting a symbol.

Receiving-end

The x86 Streaming SIMD Extension provides instructions to perform direct read and write operations to main memory without affecting the cache. A non-temporal store of double quadword from an `xmm` register into a 128-bit memory address is performed with the `movntdq` instruction [70, 58]. The receiver can use this instruction to accelerate the probing and reduce errors due to cache pollution of other processes (see Listing 2). More importantly, it prevents the cache-miss hardware performance counter from incrementing, inhibiting countermeasures based exclusively on the monitoring of cache activity. The `mfence` (lines 4 and 6) instruction plays two important roles. Firstly, it prevents re-ordering between the non-temporal store (line 5) and the reading of the counter (lines 3 and 7). Secondly, it allows flushing the write-combining (WC) buffer, thus ensuring of the execution of the non-temporal store in-order. Non-temporal operations follow WC semantics, which specify that data must not be cached so as to reduce cache pollution (i.e. when data is used only once). Non-temporal operations are combined in the WC buffer, and delayed until the buffer becomes full, or upon a serialising

event (e.g. `mfence`, `cpuid`, `lock`, etc.) [70]. We note that the size of the WC varies from one microarchitecture to another, however it can take the size of several cache lines and one single `movntdq` might not be enough to fill it up. Thus the second `mfence` instruction (line 6) ensures that the non-temporal store is not delayed until the WC buffer is full. The full source-code of the receiving-end is provided in Appendix B.

```
1  ; RSI = pointer to counter
2  ; RDI = pointer to "normal" region
3  MOV RDX, [RSI]          ; read counter value
4  MFENCE                  ; memory barrier
5  MOVNTDQ [RDI], XMM0     ; non-temporal store
6  MFENCE                  ; memory barrier
7  MOV RAX, [RSI]          ; read counter value
8  SUB RAX, RDX            ; compute elapsed time
9  RET
```

Listing 2: Receiving a symbol.

Counting-thread

In order to discard the usage of the TSC, it is replaced with a counting thread using the `inc` instruction (see Listing 3). The counter value is systematically written to memory, so as to make it visible to the receiving-end. We note that this will require the receiver to have access to a second logical CPU, and that it does not alter the resolution of measurements—in fact it can even improve it [129].

```
1  ; RDI = pointer to counter
2  XOR RBX, RBX            ; zeroise register
3  _LOOP:
4  INC RBX                 ; increment counter value
5  MOV [RDI], RBX          ; write counter value to cache
6  JMP _LOOP
```

Listing 3: Counting thread.

Algorithm 3: Memory Bus Covert Channel Protocol.

M_{S1}: exotic memory region within sender's address space;
M_{S0}: normal memory region within sender's address space;
M_R: normal memory region within receiver's address space;
N: number of bits to send;
$send[N], recv[N]$: respective buffers of sender and receiver;

Sender	**Receiver**
for all $i \in [0; N]$ **do**	**for all** $i \in [0; N]$ **do**
if $send[i] == 1$ **then**	{Timed normal access}
{Exotic access}	t = Probe(M_R);
Access(M_{S1});	**if** $t > threshold$ **then**
else	$recv[i] = 1$;
{Normal access}	**else**
Access(M_{S0});	$recv[i] = 0$;
end if	**end if**
end for	**end for**

The proposed protocol is presented in Algorithm 3. The sender performs atomic memory accesses either in a cache line-crossing region (i.e. exotic) or in a single cache line (i.e. normal), with the *Access* function referring to Listing 1. Meanwhile the receiver starts the counting thread, and probes memory accesses into a single cache line of its own userspace, with the *Probe* function referring to Listing 2.

The entire premise of the covert channel is based on the ability for the receiver to observe a timing variation depending on the sender's activity. In our AWS EC2 m5d.large instance pair (see Table 5.2), the receiver's accesses to DRAM take an average of 935 increment iterations when the sender is inactive, and 2403 increment iterations when the sender is active. Therefore, there is an average performance overhead of 1468 increment iterations per memory access. An increment iteration is the time that it takes for the counting thread to perform one increment operation (see Listing 3). On the AWS EC2 m5d.large, we measured that it takes 1498034 CPU cycles to perform 2^{20} iterations, that is an average of 1.42 CPU cycles per increment operation. This means that overhead caused by the sender's activity amounts to 2084 CPU cycles. Thus it is trivial for the receiver to differentiate the binary values sent across the covert channel.

5.4 Evaluation

This section presents the evaluation of the covert channel capacity in the AWS EC2 commercial environment. Three x86 microarchitectures are tested. This Section also discusses the effects of the attack on microarchitectural states, in regards to the requirements established in Section 5.2.

5.4.1 Experimental Setup

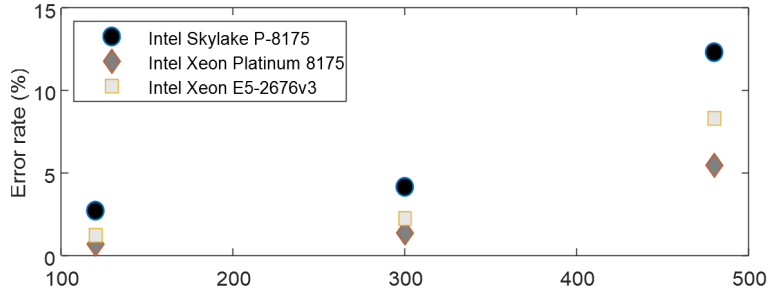
The testing environments are summarised in Table 5.2. It consists of three AWS EC2 instance pairs featuring different x86-64 microarchitectures, namely Intel Xeon E5-2676v3 (released in 2015), Intel Xeon Platinum 8175 (released in 2017), and AMD EPYC 7571 (released in 2019). The tests are repeated on each instance pair. Both the sender and the receiver run in their own instance and have access to two virtual CPUs. Furthermore, dedicated instances are used in order to ensure that sender and receiver are scheduled on the same processor.

5.4.2 Channel Capacity

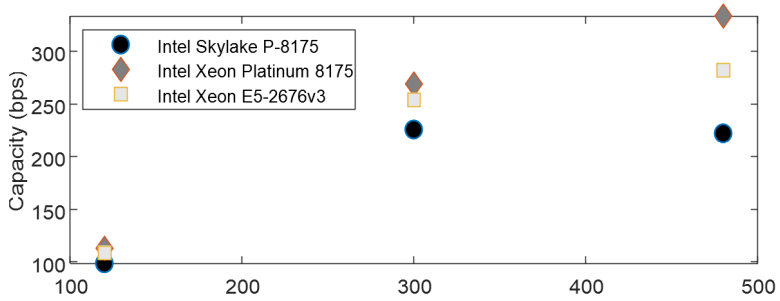
The error rate (Figure 5.1a) is computed by counting the number of bit flips over a 256-bit message. At a bitrate of 480 bps, the covert channel reaches an error rate as low as 5.46% on the Intel Xeon Platinum 8175 platform. The channel capacity (Figure 5.1b) is computed under the binary symmetric model (see Section 3.3.13). At a bitrate of 480 bps, the covert channel reaches a capacity of up to 333 bps on the Intel Xeon Platinum 8175 platform. The same order of magnitude as the original proposal is achieved [164]. Results

Table 5.2: Error rate and capacity (raw bitrate of 480 bps).

Instance type	Microarchitecture	Error rate	Capacity
m4.large	Intel Xeon E5-2676v3	8.31%	281 bps
m5a.large	AMD EPYC 7571	12.3%	221 bps
m5d.large	Intel Xeon Platinum 8175	5.46%	333 bps



(a) Error rate (%) as a function of the bit rate (bps).



(b) Channel capacity (bps) as a function of the bit rate (bps).

Figure 5.1: Error rate and capacity.

are summarised in Table 5.2. In addition, Figure 3.2 shows that this covert channel achieves the highest severity score in the CCSS framework. This is explained by its ability to circumvent countermeasures previously discussed, its high channel capacity, and its convenience to deploy in the wild.

5.4.3 Effects on Microarchitectural States

The proposed covert channel successfully meets all the design requirements previously established. First, it has been established that a high resolution timer is required in order to measure the latency of performing memory accesses. A timing channel can effectively be mitigated if the values read from this counter are too noisy. Design requirement 1 intends to prevent such countermeasure. In order to gain assurance that neither the `rdtsc` nor the `rdtscp` instructions are used, the entire binaries were disassembled in order to confirm that these instructions are not present. The receiving-end

relies exclusively on the counting thread in order to benchmark the execution of memory accesses, and there are no other benchmarking operations taking place in the code. Only the evaluation of the channel capacity requires using the TSC. Beyond the testing phase, this operation is not required.

Second, the covert channel is based on the ability for the sender and receiver to manipulate and observe microarchitectural states. In the case of cache-based covert channels, this microarchitectural state is the presence of a cache line in a cache set. Injecting noise in caches, such that the receiver and sender can lose the above-mentioned capability, can effectively mitigate the attack. Design requirement 2 intends to thwart such countermeasures. The sending-end exploits the bus locking behaviour for atomic accesses to cache-line crossing regions (see Section 5.3.1). The resulting performance cost is generated system-wide. Thus all memory accesses are impacted, whether they target caches or DRAM. Also, the receiving-end benchmarks uncached accesses only. On x86 microarchitectures, non-temporal instruction are designed to fetch data directly to DRAM. Therefore, noise injection in caches does not affect the microarchitectural state leveraged by the sender and receiver, since they only communicate via DRAM accesses. Noise injection on caches would be completely oblivious to this covert channel.

Third, software partitioning enforces spatial isolation over certain processor resources, such that co-tenants cannot share a vulnerable microarchitectural state. Design requirement 3 accounts for such countermeasures, some of which are already deployed by cloud providers. It is not possible to gain control over the disabling of SMT on the commercial platform, and dedicated instances from the same AWS account may share hardware CPUs. Therefore, an additional experiment is reproduced in a lab environment, such that SMT can be disabled and processes can be pinned to separate hardware CPUs. The covert channel was launched across two native processes on an AMD Ryzen Threadripper 1950X processor (Zen), which features 16 cores. SMT was disabled from the Basic Input Output System (BIOS menu), and each communicating was set to different cores via the (privileged) `taskset` command. This command was only used for the testing of this design requirement—it is not necessary for deploying the attack. This lab experiment shows that the

proposed covert channel allows cross-core communication, hence it cannot be mitigated on commercial platforms where SMT is disabled. We note that this lab experiment is carried out solely for the purpose of demonstrating compliance against design requirement 3, and is therefore not subject to a channel capacity evaluation. Finally, the cache architecture (e.g. set-associative) is not relevant with the memory bus. Firstly, because it does not rely on modulating a shared cache set. And secondly, because non-temporal accesses bypass the cache on x86 platforms. Therefore, cache colouring cannot have a mitigating effect on the proposed covert channel.

Finally, hardware partitioning comprises different forms of segregation which would be enforced at the hardware level, from isolated cache partitions through time-multiplexing on certain scheduling resources to privileged instructions. The tests were performed from user accounts, and the disassembling of the binaries showed that neither the `rdtsc` nor the `clflush`—theoretically privileged—are used. While the memory controller is solicited in DRAM accesses, it is not responsible for generating timing variations. Thus time-multiplexing over the memory controller or the interconnect cannot conceal timing variations caused by the bus lock behaviour. As for cache partitions, these are irrelevant with the proposed attack since timing variations do not cause any cache accesses. As a result, it can be asserted that design requirement 4 has been met.

5.5 Discussion

This section considers new approaches to mitigate the proposed covert channel, and discusses the effect of cache line-crossing atomic accesses in ARMv8.2 microarchitectures.

5.5.1 Closing the Memory Bus Covert Channel

Auditing strategies claim to have a system-wide approach to mitigating leakage channels, as other strategies allegedly focus on specific attacks. Yet, authors of these auditing mechanisms never included the bus lock counter into

their model inputs. Further work is required in order to verify whether this event influences the rate of false positives. The long-term solution to closing cross-VM covert channels could be to challenge directly the adversary model. It is assumed that each communicating entity is running in its own VM, and that the two VMs are scheduled on the same processor. A new VM allocation policy where one tenant can only be a neighbour of a small set of other would significantly reduce the chances for the attacker to achieve or maintain co-residency with the instance of the victim. This idea is first explored by Wu et al. [164], however it must be balanced with the performance cost and would only become practical as a result of a risk analysis initiated by cloud providers. Other techniques for closing the memory bus channel are discussed in Section 6.2.

Another alternative could be to develop state flushing [52] as a mean to interrupt the memory bus covert channel. State flushing consists of resetting microarchitectural states regularly during execution. For example, the x86 ISA features the `wbinvd` instruction which writes-back to system memory and invalidates cache lines of both internal and external caches. This instruction can effectively close cache side channels, but it is too aggressive to become practical—Ge et al. [51] describe it as an *overkill* and measured a performance overhead of up to 12 ms. State flushing can be considered as a side channel countermeasure, where microarchitectural states are flushed whenever a sensitive workload is executed. However it is not practical to consider state flushing as a covert channel countermeasure, since it would require flushing microarchitectural states continuously and independently of the running workload. Also, current ISAs lack support for efficient state flushing, forcing authors of these mechanisms to either develop new instructions [162], or to combine existing instructions with heavy software support [49]. Unfortunately, developing state flushing on commercial cloud platforms can only be performed by hardware manufacturers. Also, this approach would conflict with auditing techniques that already monitor abnormal microarchitectural state behaviour (e.g. against timing side channels).

A new feature known as Memory Bandwidth Allocation (MBA) has been introduced in Intel Xeon Scalable processors [67]. This feature allows con-

trolling the memory bandwidth of each core, and could be leveraged in order to inhibit the memory bus covert channel. The advantage of this approach is that it relies on existing hardware support, much like Liu et al. [90] used the Intel CAT feature in order to close Prime+Probe cache attacks. Note that Intel CAT was also used by Lipp et al. [87] as a Rowhammer enhancer [79], who then suggested to modify Intel CAT in order to mitigate the vulnerability. It is possible that Intel MBA could also lead to new vulnerabilities, as many other hardware optimization techniques did, e.g. out-of-order execution, SMT, prefetching, etc. Finally, Intel MBA is not available on all Intel microarchitectures.

5.5.2 The Case of ARMv8.2-A

ARM processors have recently arrived on cloud platforms with the Neoverse microarchitectures. Thus it is expected that the share of x86 processors in IaaS will decrease for the benefit of ARM architectures. Reproducing the memory bus covert channel across two AWS EC2 instances platform featuring a 64-bit ARM architecture was not successful. It was possible to re-create the receiving-end, the only exception being that non-temporal instructions might not be guaranteed to be served from DRAM. However, the sending-end, which uses the A64 `swp` instruction—equivalent of the x86 `xchg`—is deprecated since the ARMv6 ISA. To the best of my knowledge, this instruction should be re-introduced in the ARMv8.2-A ISA or upcoming versions, but it will no longer generate the desired system-wide "bus lock". Its behaviour might be similar to the load-acquire store-release mechanism. Therefore, it has not been possible to reproduce the timing channel on the Graviton and Graviton2 processors.

5.6 Summary

This chapter analysed the set of potential countermeasures against microarchitectural covert channel attacks in IaaS environments. A new instance of the memory bus lock exploitation was then created, where all defence mech-

anisms become obsolete. The proposed attack was tested in a the AWS EC2 commercial environment. The attack reached an effective capacity of 333 bps while remaining effective against noise injection, software partitioning, and hardware partitioning defence strategies. The objective of this chapter was to close the gap between newly released attacks and countermeasures. Despite an extensive literature review, this chapter has shown that there are so far no means of addressing the memory bus lock vulnerability. It is hoped that this work will stimulate the community to investigate system-wide countermeasures against timing channels rather than incremental solutions. A discussion on how to close the proposed covert channel was finally provided, along with a comparison against the ARMv8.2-A architecture which, at the time of this writing, is only emerging on the IaaS market. The results of this work have been reported to AWS, Intel, and AMD for responsible disclosure.

Chapter 6

Conclusion

Contents

This chapter summarises the contributions of this thesis and discusses new research directions in line with multitenant computing environment security.

6.1 Summary and Conclusions

Microarchitectural timing-based attacks are software-launched exploits that leverage the sharing of processing resource among multiple tenants, in order to compromise sensitive information. These attacks can either take the form of a side channel, where the victim is accidentally leaking information, or the form of a covert channel, where the attacker has infected the victim with a malicious sending-end that deliberately transmits information. These timing-based attacks have been increasingly popular in the last decade, and even more since the release of the Spectre and Meltdown exploits [81, 89]. Despite the exposure of shared computing environments to such attacks, cloud providers keep on attracting more and more businesses willing to outsource their infrastructure. In this context, covert channels become particularly interesting for leaking information in a non-conspicuous manner, e.g. to avoid generating network traffic and associated logs [135]. They are relevant with advanced persistent threats, where an attacker employs cutting-edge techniques in order to maintain long-term intrusion and data exfiltration capabilities. As a result, a covert channel is ideal for stealthy data theft on high-profile targets.

In this thesis, an evaluation was first conducted on the severity of covert channel attacks against Infrastructure-as-a-Service. Using the Common Vulnerability Scoring System (CVSS v3.1), medium severity scores of up to 5.0 were obtained. In comparison, the MySQL Stored SQL Injection vulnerability (CVE-2013-0375) also achieved a medium severity score, and was patched shortly after its disclosure. To this day, there are still no practical countermeasures against practical covert channel attacks released several years ago [122, 164]. This study reveals that these attacks are not theoretical threats, and that they require the immediate attention of the community. A new framework for evaluating microarchitectural covert channels was then proposed. It allows comparing attacks based on their performance as well as the associated adversary model, and successfully outlines shortcomings which the CVSS fails to identify. For example, it was shown that certain cache-based attacks would obtain similar CVSS scores as other attacks despite substan-

tial limitations in their adversary model. The proposed framework puts in evidence these disparities, thanks to additional metrics and a bespoke scoring system. This research thread managed to identify the conditions for circumventing contemporary cloud environment's information flow policies. Resulting vulnerabilities can in turn be exploited, as demonstrated hereafter.

Secondly, this thesis presents a new covert channel attack based on the modulation of Intel integrated memory controllers, with two possible variants. The first attack is privileged and is to be deployed in native environments. Test results demonstrated a capacity of up to 729 bps. The second attack is unprivileged and is to be deployed in virtualized environments, with a capacity of up to 95 bps. The memory controller-based attack allows establishing a rogue communication channel across processes running on separate processor cores, thus circumventing defense strategies based on the disabling of SMT. In addition from being resilient to software partitioning and other noise injection countermeasures, this technique facilitates the deployment of covert channels by eliminating the negotiation phase between communicating entities—such as finding congruent addresses in Prime+Probe attacks [116]—and does not rely on the availability of shared memory—such as in Flush+Reload attacks [56]. As such, it provides a viable alternative to existing works as well as new challenges for hardware manufacturers. Despite expanding the range of possible attack vectors, laboratory experiments also demonstrated the effect of virtualisation on cross-VM covert channels. In particular, it was shown that overcoming the addressing uncertainty while holding out against background noise is not trivial. This consideration forces devising even further innovative methods in order to propose reliable communication while challenging existing and potential countermeasures.

Thirdly, a comprehensive analysis of state-of-the-art countermeasures was conducted, including but not limited to noise injection, software partitioning, and hardware partitioning techniques. It was shown that the strategies proposed in academia consist of incremental changes to the hardware architecture, operating system, or hypervisor. In parallel, most of the new leakage channel designs aimed at exploiting residual vulnerabilities, without challenging latest developments in terms of defences. Using the findings of

this analysis, a new covert channel was devised that leverages the x86 bus lock vulnerability. The resulting attack allows cross-core, cross-VM attacks with a channel capacity of up to 333 bps. A rogue channel could be established and tested within a commercial environment between two AWS EC2 instances, thus asserting of the applicability of such attacks in the wild. Tests targeted several x86 and ARMv8 microarchitectures released between 2015 and 2019. This new design disregards common issues such as addressing uncertainty and enables high-speed communication channels in a virtualized environment. However, its main contribution lies in its ability to overcome the large set of defences published in academia, whether these are theoretical or already deployed. This research thread thus demonstrates that outsourcing data to a third-party cloud provider presents a risk, and that a motivated attacker can easily make his way around information flow policies in order to leak this data without being detected.

The research publications associated to Chapter 4 and Chapter 5 has lead to a responsible disclosure to Intel, AMD, and AWS, regarding vulnerabilities to timing channels of the integrated memory controller and the memory bus. Intel has assured that such vulnerability was covered by a publicly available guidance [68]. AWS responded that this research item constitutes "a further reminder that customers should not run un-trusted code on their instances to fulfill the customer side of the shared responsibility model". Besides responsible disclosures, the work presented in this thesis has lead to new publications: covert channels targeting ARM-based Android devices [136], and fingerprinting of sensitive workloads on ARM, x86, and RISC-V implementations [136, 137].

6.2 Recommendations for Future Work

This thesis aims at studying microarchitectural covert channel vulnerabilities in multi-tenant environments, from the adversary model through new attack mechanisms, to a study of defense strategies. The resulting analysis opens new research directions, along with other open challenges such as new attack vectors.

For instance, a Javascript-based covert channel would enable the attacker to embed the sending-end into a malicious browser or website plugin. As a result, the adversary would be able to penetrate the victim's environment at runtime. Javascript-based microarchitectural side-channel attacks have been successfully demonstrated in the past [55, 115]. Following their publication, access to high-resolution timers from web-based scripts has been restricted. However, the sending-end does not require to probe any memory access. It would be worth investigating which shared resource, aside of the last-level cache, can be modulated accurately in Javascript in order to create a reliable covert channel across VMs.

The malicious payload could also be concealed in an existing program by means of return-oriented programming (ROP). ROP allows an attacker who has hijacked a program's control flow to craft a rogue software payload by re-using chunks of the victim program [29, 125]. Crafting an encoding an artifact in this way would discard the necessity to rely on a standalone malicious software, thus improving the attacker's ability to maintain persistent access while remaining undetected. This approach could enable an attacker circumventing countermeasures based on the fingerprinting of malicious workloads, e.g. signature-based malware detection. Further work is required to determine whether this approach could contribute to reducing the complexity of deploying covert channel attacks.

Alternatively, auditing strategies have been proposed in academia to mitigate side channel vulnerabilities [175, 9, 31, 84]. Among others, Zhang et al. [178] developed a technique that enables VMs to detect timing-based side channel on the LLC. The victim continuously probes memory accesses to detect anomalies. Yet, this approach might incur a significant performance cost. Auditing strategies claim to have a system-wide approach to mitigating leakage channels. Authors of these auditing mechanisms usually omit the bus lock counter from their model inputs. This metric measures the ratio of bus cycles, during which a `LOCK#` signal is asserted on the bus. It would be trivial to account for this metric during anomaly detection, however its effect on false positives remains an open question.

Another approach consist in enforcing static analysis at intermediate

stages of the software supply chain. Recent events have shown that the complexity of this ecosystem benefits to malicious individuals [112]. Static code analysis cannot be enforced on the complete cloud image. Moreover, while OS providers might apply good security practices, it is not necessarily the case for all user application. Further work is required in order to analyse the effect of implementing control checks at different stages of the development process. It is therefore necessary to identify the key stages where security evaluations may be required in order to detect malicious payloads.

The long-term solution to closing cross-VM covert channels could be to challenge directly the adversary model, which assumes that both the attacker's and victim's virtual machines are co-located. Wu et al. [164] suggest an approach to mitigating covert channels in public clouds where the policy requires each tenant to neighbour only one other tenant. This is a trade-off between dedicated instances [134] and complete sharing of computing resources. This approach renders covert channel attacks almost impractical, however the operational cost remains an open-question.

Appendix A

Memory Bus-based Covert Channel Sender

```
#include <iostream>
#include <cstdint>
#include <cstdlib>
#include <cassert>
#include <time.h>
#include <unistd.h>
#include <vector>
#include <sys/time.h>
#include <sys/mman.h>

using namespace std;

#define LOGERROR(f, ...)                              \
        do{ printf("[%-5s] ", "ERROR");               \
        printf(f, __VA_ARGS__); exit(EXIT_FAILURE); \
        }while (0)
#define LOGINFO(f, ...)                               \
        do{ printf("[%-5s] ", "INFO");                \
        printf(f, __VA_ARGS__); }while (0);
#define MMAP_PROT  PROT_READ | PROT_WRITE
#define MMAP_FLAGS MAP_POPULATE | MAP_ANONYMOUS | MAP_PRIVATE |
↪  MAP_LOCKED

/* ********************************************** */
```

```
24
25 extern "C" uint16_t asm_snd(uint64_t*);
26 extern "C" uint16_t asm_nop(uint64_t*);
27
28 /* ********************************************** */
29
30 long int timerset(int off){
31 timeval t0;
32 if (gettimeofday(&t0, NULL))
33         LOGERROR("%s", "gettimeofday\n");
34 return (t0.tv_sec - (t0.tv_sec)%100 + (long int) off);
35 }
36
37 /* ********************************************** */
38
39 int main(int argc, char *argv[]){
40 cout << endl;
41 assert( argc==2 );
42
43 // Initialize message
44 uint8_t msg[256] = {
45 0,1,0,0,1,0,1,1,0,1,0,0,1,1,1,0,0,0,1,1,0,1,1,1,0,1,0,1,0,0,1,0,
46 0,1,0,1,0,1,1,0,0,1,0,0,0,0,1,1,0,1,1,1,1,1,1,0,1,0,0,1,0,1,1,0,0,
47 0,1,0,0,1,0,0,0,0,0,1,1,0,1,1,0,0,0,1,0,1,1,1,0,0,1,0,1,1,1,1,1,0,
48 0,1,0,0,0,1,0,1,0,0,1,0,0,1,0,1,0,1,0,0,1,1,0,1,0,1,0,0,0,1,1,0,
49 0,0,1,1,0,0,1,0,0,0,1,0,1,1,1,0,0,1,0,1,1,0,0,0,0,1,0,1,0,0,0,1,
50 0,1,1,0,0,1,1,0,0,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,0,1,0,0,1,1,0,0,
51 0,0,1,1,0,1,1,1,0,1,1,0,0,0,1,0,0,0,1,0,1,0,1,0,0,1,1,0,1,0,0,0,
52 0,1,0,0,0,0,1,0,0,0,1,1,1,0,0,1,0,1,1,0,0,1,0,0,0,1,0,1,1,1,1,1};
53
54 // Initialize bit rate
55 size_t M1 = 2400;
56 size_t M0 = M1*1.27;
57
58 // Initialize pointer
59 void* ptr0 = mmap(0, getpagesize(), MMAP_PROT, MMAP_FLAGS, -1, 0);
60 if (ptr0==(void*)-1)
61         LOGERROR("%s", "mmap\n");
62 if (mlock(ptr0, getpagesize()))
```

```
63          LOGERROR("%s", "mlock\n");
64  assert( (uintptr_t)ptr0%0x40==0 );
65  void* ptr1 = reinterpret_cast<void*>(reinterpret_cast<uint8_t*>(ptr0)
    ↪  +63*sizeof(uint8_t));
66
67  // Initialize countdown
68  long int t1 = timerset(atoi(argv[1]));
69  LOGINFO("%s", "counting down...\n");
70  timeval t0;
71  while (1) {
72          if (gettimeofday(&t0, NULL))
73                  LOGERROR("%s", "gettimeofday");
74          if (!(t1-(t0.tv_sec)))
75                  break;
76  }
77  /** NO CODE HERE **/
78  // Transmit message
79  for (size_t i=0; i<256; ++i) {
80          if (msg[i]==1){
81                  for (size_t j=0; j<M1; j++)
82                          asm_snd((uint64_t*)ptr1);
83          } else {
84                  for (size_t j=0; j<M0; j++)
85                          asm_nop((uint64_t*)ptr0);
86          }
87  }
88
89  LOGINFO("%s", "sender has finished\n");
90  cout << endl;
91  return 0;
92  }
```

Appendix B

Memory Bus-based Covert Channel Receiver

```
#include <algorithm>
#include <iterator>
#include <iostream>
#include <cstdint>
#include <cstdlib>
#include <cassert>
#include <vector>
#include <array>
#include <cmath>
#include <cstring>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/mman.h>

using namespace std;

#define LOGERROR(f, ...)                          \
        do{ printf("[%-5s] ", "ERROR");           \
        printf(f, __VA_ARGS__); exit(EXIT_FAILURE); \
        }while (0)
#define LOGINFO(f, ...)                           \
        do{ printf("[%-5s] ", "INFO");            \
        printf(f, __VA_ARGS__); }while (0);
```

```
25 #define MMAP_PROT  PROT_READ | PROT_WRITE
26 #define MMAP_FLAGS MAP_POPULATE | MAP_ANONYMOUS | MAP_PRIVATE |
   ↪  MAP_LOCKED
27
28 /* ********************************************** */
29
30 extern "C" uint16_t asm_rcv(uint64_t*, uint64_t*);
31 extern "C" void asm_clk(uint64_t*);
32 uint64_t clk;
33
34 /* ********************************************** */
35
36 long int timerset(int off)
37 {
38 timeval t0;
39 if (gettimeofday(&t0, NULL))
40         LOGERROR("%s", "gettimeofday\n");
41 return (t0.tv_sec - (t0.tv_sec)%100 + (long int) off);
42 }
43
44 /* ********************************************** */
45
46 void record(const vector<uint16_t> vec, const char *path)
47 {
48 remove(path);
49 size_t pos (0);
50 FILE *f;
51 f = fopen(path, "a");
52 for (auto it=vec.begin(); it!=vec.end(); it++) {
53         char s_buf[256];
54         for (size_t j=0; j<sizeof s_buf; j++)
55                 s_buf[j] = 0;
56         snprintf(s_buf, sizeof s_buf, "%lu\t%u\n", pos, *it);
57         if (f==NULL)
58                 LOGERROR("%s", "fopen");
59         size_t k = 0;
60         for (k=0, fputc(s_buf[k], f);
           ↪  s_buf[k]!='\n';++k,fputc(s_buf[k], f))
61                 ;
```

```
62          pos++;
63          }
64  fclose(f);
65  }
66
67  /* ************************************************ */
68
69  void prune(vector<uint16_t>& vec1)
70  {
71  vector<uint16_t> vec2;
72  vec2.resize(floor(vec1.size()/2));
73  for (size_t i=0; i<vec2.size(); i++) {
74          if (i==0) {
75                  vec2[i] = (vec1[i] + vec1[2*i])/2;
76          } else {
77                  vec2[i] = (vec1[2*i-1] + vec1[2*i])/2;
78          }
79  }
80  vec1.clear();
81  vec1 = vec2;
82  }
83
84  /* ************************************************ */
85
86  void check(const array<uint8_t, 256> arr1)
87  {
88  const array<uint8_t, 256> arr2 = {
89  0,1,0,0,1,0,1,1,0,1,0,0,1,1,1,0,0,0,1,1,0,1,1,1,0,1,0,1,0,0,1,0,
90  0,1,0,1,0,1,1,0,0,1,0,0,0,0,1,1,0,1,1,1,1,1,1,0,1,0,0,1,0,1,1,0,0,
91  0,1,0,0,1,0,0,0,0,0,1,1,0,1,1,0,0,0,1,0,1,1,1,0,0,1,0,1,1,1,1,1,0,
92  0,1,0,0,0,1,0,1,0,0,1,0,0,1,0,1,0,1,0,0,1,1,0,1,0,1,0,0,0,1,1,0,
93  0,0,1,1,0,0,1,0,0,0,1,0,1,1,1,0,0,1,0,1,1,0,0,0,0,1,0,1,0,0,0,1,
94  0,1,1,0,0,1,1,0,0,1,0,0,0,0,1,1,0,1,1,0,0,0,0,1,0,1,0,0,1,1,0,0,
95  0,0,1,1,0,1,1,1,0,1,1,0,0,0,1,0,0,0,1,0,1,0,1,0,0,1,1,0,1,0,0,0,
96  0,1,0,0,0,0,0,1,0,0,0,1,1,1,0,0,1,0,1,1,0,0,1,0,0,0,1,0,1,1,1,1,1};
97
98  uint8_t errors (0);
99  for (size_t i=0; i<256; i++) {
100         if (arr1[i]!=arr2[i])
```

```
101                 errors++;
102 }
103
104 LOGINFO("message received (%d errors)\n", errors);
105
106 if (errors) {
107         putchar('\n');
108         for (size_t i=0; i<256; i++) {
109                 if (arr1[i]!=arr2[i]) {
110                         printf("\033[91m%d\033[m", arr1[i]);
111                 } else {
112                         printf("%d", arr1[i]);
113                 }
114                 if ((i+1)%8==0)
115                         putchar(' ');
116                 if ((i+1)%32==0)
117                         putchar('\n');
118         }
119         putchar('\n');
120 }
121 }
122
123 /* ************************************************ */
124
125 void read(vector<uint16_t> vec1, int64_t raw_avg)
126 {
127 // Assign binary value to each sample
128 vector<uint16_t> vec2;
129 vec2.resize(vec1.size());
130 for (size_t i=0; i<vec1.size(); i++) {
131         if (vec1[i]<=raw_avg) {
132                 vec2[i] = 0;
133         } else {
134                 vec2[i] = 1;
135         }
136 }
137
138 // Find index of last effective sample
139 size_t off;
```

```
140 vector<uint16_t> foo{1,1,1,1,1,1,1,1};
141 vector<uint16_t>::iterator itr;
142 itr = find_end(vec2.begin(), vec2.end(), foo.begin(), foo.end());
143 if (itr==vec2.end()) {
144         LOGINFO("%s", "failed to find pattern\n");
145         return;
146 } else {
147         off = distance(vec2.begin(), itr);
148 }
149
150 // Remove non-effective samples
151 vec1.erase(vec1.begin()+off+foo.size(), vec1.end());
152 vec1.shrink_to_fit();
153
154 // Compute upper/lower bound latency
155 int64_t max_sum (0), min_sum (0), max_count (0), min_count (0),
    ↪  max_avg (0), min_avg (0);
156 for (size_t i=0; i<vec1.size(); i++) {
157         if (vec1[i]<=raw_avg) {
158                 min_sum += vec1[i];
159                 min_count++;
160         } else {
161                 max_sum += vec1[i];
162                 max_count++;
163         }
164 }
165 max_avg = max_sum/max_count;
166 min_avg = min_sum/min_count;
167 LOGINFO("uper bound is %lu cycles\n", max_avg);
168 LOGINFO("lower bound is %lu cycles\n", min_avg);
169
170 // Remove outliers
171 vector<uint16_t> vec3;
172 for (size_t i=0; i<vec1.size(); i++) {
173         if (vec1[i]<=(max_avg+max_avg*0.05))
174                 vec3.push_back(vec1[i]);
175         else
176                 vec3.push_back(max_avg);
177 }
```

```
178
179 vec1.clear();
180 vec1 = vec3;
181
182 // Decode measurements
183 size_t len (256), k (0);
184 float k_float (0);
185 float step = (float)vec1.size()/(float)len;
186 array<uint8_t, 256> arr;
187 while (len--) {
188         vector<uint16_t> vec4;
189         int64_t sum (0), avg (0);
190         for (size_t j=0; j<floor(step); j++)
191                 vec4.push_back(vec1[j+k]);
192         for (auto it=vec4.begin(); it!=vec4.end(); it++)
193                 sum += *it;
194         avg = sum / vec4.size();
195         arr[255-len] = (avg>raw_avg) ? 1 : 0;
196         k_float += step;
197         k = floor(k_float);
198 }
199
200 // Count errors
201 check(arr);
202 }
203
204 /* ********************************************** */
205
206 void* counter(void* ptr)
207 {
208 asm_clk(&clk);
209 pthread_exit(NULL);
210 }
211
212 /* ********************************************** */
213
214 int main(int argc, char* argv[])
215 {
216 cout << endl;
```

```
217 assert( argc==2 );
218
219 // Initializae measurement vector
220 vector<uint16_t> vec1;
221 vec1.resize(8388608); // 2^23
222
223 uint32_t ts0h (0), ts0l (0), ts1h (0), ts1l (0);
224 uint64_t ela (0);
225
226 // Initialize pointer
227 void* ptr = mmap(0, getpagesize(), MMAP_PROT, MMAP_FLAGS, -1, 0);
228 if (ptr==(void*)-1)
229         LOGERROR("%s", "mmap\n");
230 if (mlock(ptr, getpagesize()))
231         LOGERROR("%s", "mlock\n");
232
233 // Initialize counting thread
234 clk = 0;
235 pthread_t thread;
236 if (pthread_create(&thread, NULL, counter, NULL)!=0)
237         LOGERROR("%s", "pthread_create\n");
238
239 // Initialize countdown
240 long int t1 = timerset(atoi(argv[1]));
241 LOGINFO("%s", "counting down...\n");
242 timeval t0;
243 while (1) {
244         if (gettimeofday(&t0, NULL))
245                 LOGERROR("%s", "gettimeofday");
246         if (!(t1-(t0.tv_sec)))
247                 break;
248 }
249
250 // Start bench-marking
251 __asm volatile("cpuid\n" : : : "rax","rbx","rcx","rdx");
252 __asm volatile("rdtsc\n\t                \
253                 mov %%edx, %0\n\t        \
254                 mov %%eax, %1\n\t" : "=r"(ts0h),"=r"(ts0l) : :
↪  "rax","rdx");
```

```
255
256 /** NO CODE HERE **/
257 // Measure access latency to pointer
258 for (auto &n: vec1)
259         n = (uint16_t) (asm_rcv((uint64_t*)ptr, &clk) & 0xFFFF);
260
261 // Stop bench-marking
262 __asm volatile("rdtscp\n\t                \
263                 mov %%edx, %0\n\t         \
264                 mov %%eax, %1\n\t" : "=r"(ts1h),"=r"(ts1l) : :
    ↪  "rax","rdx");
265 __asm volatile("cpuid\n" : : : "rax","rbx","rcx","rdx");
266 ela = (((uint64_t) ts1h << 32) | ts1l) - (((uint64_t) ts0h << 32) |
    ↪  ts0l);
267 LOGINFO("bit rate %f bit/s\n", ( 1 / ( (float)( (float)ela / 2.5 ) /
    ↪  256 ) ) * 1000000000);
268
269 // Prune measurements
270 vector<uint16_t> vec2 = vec1;
271 for (size_t i=0; i<10; i++)
272         prune(vec2);
273
274 // Compute average
275 int64_t sum (0), avg (0);
276 for (auto it=vec2.begin(); it!=vec2.end(); it++)
277         sum += *it;
278 avg = sum / vec2.size();
279 LOGINFO("average access time is %lu cycles\n", avg);
280
281 // Decode measurements
282 read(vec2, avg);
283
284 // Record pruned measurements
285 record(vec2, "sim/measures.dat");
286 LOGINFO("%s", "measurements recorded\n");
287
288 LOGINFO("%s", "receiver has finished\n");
289 cout << endl;
290 return 0;
```

```
291 }
```

Bibliography

[1] Onur Aciiçmez. Yet another microarchitectural attack: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.

[2] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124. Springer, 2010.

[3] 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*, pages 312–320, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, pages 225–242. Springer, 2007.

[5] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Cryptographers' Track at the RSA Conference*, pages 256–273. Springer, 2008.

[6] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the AES. In *Cryptographers' track at the RSA conference*, pages 271–286. Springer, 2007.

[7] Ayaz Akram, Maria Mushtaq, Muhammad Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. Meet the Sherlock Holmes' of side channel leakage: a survey of cache SCA detection techniques. *IEEE Access*, 8:70836–70860, 2020.

[8] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Sourangshu Bhattacharya. Performance counters to rescue: a machine learning based safeguard against micro-architectural side-channel-attacks. *IACR Cryptol. ePrint Arch.*, 2017:564, 2017.

[9] Zirak Allaf, Mo Adda, and Alexander Gegov. A comparison study on Flush+Reload and Prime+Probe attacks on AES using machine learning approaches. In *UK Workshop on Computational Intelligence*, pages 203–213. Springer, 2017.

[10] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 422–435, 2016.

[11] Cloud Security Alliance. The notorious nine: cloud computing top threats in 2013. *Top Threats Working Group*, pages 1–10, 2013.

[12] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy*, pages 623–639. IEEE, 2015.

[13] Shahid Anwar, Zakira Inayat, Mohamad Fadli Zolkipli, Jasni Mohamad Zain, Abdullah Gani, Nor Badrul Anuar, Muhammad Khurram Khan, and Victor Chang. Cross-VM cache-based side channel attacks and proposed prevention mechanisms: a survey. *Journal of Network and Computer Applications*, 93:259–279, 2017.

[14] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa M Marvel. Catch

me if you can: a closer look at malicious co-residency on the cloud. *IEEE/ACM Transactions on Networking*, 27(2):560–576, 2019.

[15] Roberto Maria Avanzi. Side channel attacks on implementations of curve-based cryptographic primitives. *IACR Cryptol. ePrint Arch.*, 2005:17, 2005.

[16] R Barona and EA Mary Anita. A survey on data breach challenges in cloud computing security: issues and threats. In *2017 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, pages 1–8. IEEE, 2017.

[17] VMware Knowledge Base. Security considerations and disallowing inter-virtual machine transparent page sharing. `https://kb.vmware.com/s/article/2080735`. Last accessed 29 Jul 2021.

[18] Naomi Benger, Joop Van de Pol, Nigel P Smart, and Yuval Yarom. "Ooh aah... just a little bit": a small amount of side channel can go a long way. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 75–92. Springer, 2014.

[19] Daniel J Bernstein. Cache-timing attacks on AES. 2005.

[20] Johann Betz, Dirk Westhoff, and Günter Müller. Survey on covert channels in virtual machines and cloud computing. *Transactions on Emerging Telecommunications Technologies*, 28(6):e3134, 2017.

[21] Arnab Kumar Biswas, Dipak Ghosal, and Shishir Nagaraja. A survey of timing channels and countermeasures. *ACM Computing Surveys*, 50(1):1–39, 2017.

[22] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 201–215. Springer, 2006.

[23] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: detecting cache attacks through self-observation.

In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, pages 224–235, 2018.

[24] Billy Bob Brumley and Risto M Hakala. Cache-timing template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 667–684. Springer, 2009.

[25] Yuriy Bulygin. CPU side-channels vs. virtualization malware: the good, the bad or the ugly. *ToorCon: Seattle, Seattle, WA, US*, 2008.

[26] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: leaking data on meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019.

[27] Carlos Cardenas and Rajendra V Boppana. Detection and mitigation of performance attacks in multi-tenant cloud computing. In *1st International IBM Cloud Academy Conference, Research Triangle Park, NC, US*, page 48, 2012.

[28] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy*, pages 1416–1432. IEEE, 2020.

[29] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[30] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: stealing intel secrets from SGX enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy*, pages 142–157. IEEE, 2019.

[31] Jie Chen and Guru Venkataramani. CC-hunter: uncovering covert timing channels on shared processor hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 216–228. IEEE, 2014.

[32] Google Cloud. Sole-tenant nodes. `https://cloud.google.com/compute/docs/nodes`. Last accessed 29 Jul 2021.

[33] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: an empirical study of timing channels on sel4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 570–581, 2014.

[34] Thomas M. Cover and Joy A. Thomas. Information theory and statistics. *Elements of Information Theory*, 1(1):279–335, 1991.

[35] Robert J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[36] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: efficiently recovering long-term secrets of SGX EPID via cache attacks. 2018.

[37] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: low-complexity mitigation of cache side channel attacks. *ACM TACO*, 8(4):1–21, 2012.

[38] Nadia El Mrabet, Jacques JA Fournier, Louis Goubin, and Ronan Lashermes. A survey of fault attacks in pairing based cryptography. *Cryptography and Communications*, 7(1):185–205, 2015.

[39] Eurostat. Cloud computing - statistics on the use by enterprises. `https://ec.europa.eu/eurostat/statistics-explained/index.php/Cloud_computing_-_statistics_on_the_use_by_enterprises`. Last accessed 29 Jul 2021.

[40] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–13. IEEE, 2016.

[41] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: a new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 53(2):693–707, 2018.

[42] Junfeng Fan, Xu Guo, Elke De Mulder, Patrick Schaumont, Bart Preneel, and Ingrid Verbauwhede. State-of-the-art of secure ECC implementations: a survey on known side-channel attacks and countermeasures. In *2010 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 76–87. IEEE, 2010.

[43] Junfeng Fan and Ingrid Verbauwhede. An updated survey on secure ECC implementations: attacks, countermeasures and cost. In *Cryptography and Security: From Theory to Applications*, pages 265–282. Springer, 2012.

[44] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Prodact: prefetch-obfuscator to defend against cache timing channels. *International Journal of Parallel Programming*, 47(4):571–594, 2019.

[45] 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*, pages 1–12, 2015.

[46] Gartner. Gartner forecasts worldwide public cloud revenue to grow 6.3% in 2020. `https://www.gartner.com/en/newsroom/press-releases/2020-07-23-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-6point3-percent-in-2020`. Last accessed 29 Jul 2021.

[47] Gartner. Gartner says worldwide IaaS public cloud services market grew 37.3% in 2019. `https://www.gartner.com/en/newsroom/press-releases/2020-08-10-gartner-says-worldwide-iaas-public-cloud-services-market-grew-37-point-3-percent-in-2019`. Last accessed 29 Jul 2021.

[48] Gartner. Gartner says worldwide IaaS public cloud services market grew 40.7% in 2020. `https://www.gartner.com/en/newsroom/press-releases/2021-06-28-gartner-says-worldwide-iaas-public-cloud-services-market-grew-40-7-percent-in-2020`. Last accessed 29 Jul 2021.

[49] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing os abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[50] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[51] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: we need a new hardware-software contract. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–9, 2018.

[52] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 163–170. IEEE, 2013.

[53] Michael Godfrey and Mohammad Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE Transactions on Cloud Computing*, 2(4):395–408, 2014.

[54] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[55] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer js: a remote software-induced fault attack in javascript. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 300–321. Springer, 2016.

[56] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[57] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[58] Mordechai Guri, Assaf Kachlon, Ofer Hasson, Gabi Kedma, Yisroel Mirsky, and Yuval Elovici. GSMem: data exfiltration from air-gapped computers over GSM frequencies. In *24th USENIX Security Symposium*, pages 849–864, 2015.

[59] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: abusing legacy x86 memory segmentation to spy on enclaved execution. In *International Symposium on Engineering Secure Software and Systems*, pages 44–60. Springer, 2018.

[60] Amir Herzberg, Haya Shulman, Johanna Ullrich, and Edgar Weippl. Cloudoscopy: services discovery and topology mapping. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, pages 113–122, 2013.

[61] Wei-Ming Hu. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, page 52. IEEE, 1992.

[62] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3-4):233–254, 1992.

[63] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[64] IDC. Worldwide server market revenue declined 6.0% year over year in the first quarter of 2020, according to IDC. `https://www.idc.com/getdoc.jsp?containerId=prUS46534520`. Last accessed 29 Jul 2021.

[65] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 368–388. Springer, 2016.

[66] Intel. Intel Transactional Synchronization Extensions (Intel TSX) Overview. `https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intel-transactional-synchronization-extensions-intel-tsx-overview.html`. Last accessed 29 Jul 2021.

[67] Intel. Introduction to memory bandwidth allocation. `https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html`. Last accessed 29 Jul 2021.

[68] Intel. Security best practices for side channel resistance. `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/security-best-practices-side-channel-resistance.html`. Last accessed 29 Jul 2021.

[69] Intel. Improving real-time performance by utilizing cache allocation technology, 2015.

[70] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, 2016.

[71] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, 2019.

[72] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain cross-VM attacks on Xen and VMware. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 737–744. IEEE, 2014.

[73] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[74] Jay Jacobs, Sasha Romanosky, Benjamin Edwards, Idris Adjerid, and Michael Roytman. Exploit prediction scoring system (EPSS). *Digital Threats: Research and Practice*, 2(3):1–17, 2021.

[75] Yier Jin. Introduction to hardware security. *Electronics*, 4(4):763–784, 2015.

[76] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[77] Rafiullah Khan, Kieran McLaughlin, David Laverty, and Sakir Sezer. STRIDE-based threat modeling for cyber-physical systems. In *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe*, pages 1–6. IEEE, 2017.

[78] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTH-MEM: system-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium*, pages 189–204, 2012.

[79] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[80] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):1–70, 2014.

[81] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE, 2019.

[82] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Hardware-software integrated approaches to defend against software cache-based side channel attacks. In *2009 IEEE 15th international symposium on high performance computer architecture*, pages 393–404. IEEE, 2009.

[83] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies*, 2018.

[84] Yusuf Kulah, Berkay Dincer, Cemal Yilmaz, and Erkay Savas. SpyDetector: an approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, 18(4):393–422, 2019.

[85] Ralph Langner. Stuxnet: dissecting a cyberwarfare weapon. *IEEE Security and Privacy*, 9(3):49–51, 2011.

[86] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX

enclaves with branch shadowing. In *26th USENIX Security Symposium*, pages 557–574, 2017.

[87] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: inducing rowhammer faults through network requests. *arXiv preprint arXiv:1805.04956*, 2018.

[88] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: cache attacks on mobile devices. In *25th USENIX Security Symposium*, pages 549–564, 2016.

[89] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[90] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture*, pages 406–418. IEEE, 2016.

[91] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 203–215. IEEE, 2014.

[92] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[93] Weijie Liu, Debin Gao, and Michael K Reiter. On-demand time blurring to support side-channel defense. In *European Symposium on Research in Computer Security*, pages 210–228. Springer, 2017.

[94] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks and defenses in cryptography. *arXiv preprint arXiv:2103.14244*, 2021.

[95] Yangdi Lyu and Prabhat Mishra. A survey of side-channel attacks on caches and countermeasures. *Journal of Hardware and Systems Security*, 2(1):33–50, 2018.

[96] Kernel Virtual Machine. Host support status. `https://www.linux-kvm.org/page/Host_Support_Status`. Last accessed 29 Jul 2021.

[97] Giorgi Maisuradze and Christian Rossow. ret2spec: speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.

[98] Andrew Marshall, Michael Howard, Grant Bugher, Brian Harden, Charlie Kaufman, Martin Rues, and Vittorio Bertocci. Security best practices for developing windows azure applications. *Microsoft Corp*, page 42, 2010.

[99] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *2012 39th Annual International Symposium on Computer Architecture*, pages 118–129. IEEE, 2012.

[100] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 46–64. Springer, 2015.

[101] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*, volume 17, pages 8–11, 2017.

[102] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. 2011.

[103] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: how SGX amplifies the power of cache attacks. In *Interna-*

tional Conference on Cryptographic Hardware and Embedded Systems, pages 69–90. Springer, 2017.

[104] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: a false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47(4):538–570, 2019.

[105] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: from general purpose to a proof of information flow enforcement. In *2013 IEEE Symposium on Security and Privacy*, pages 415–429. IEEE, 2013.

[106] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Muneeb Yousaf, Umer Farooq, Vianney Lapotre, and Guy Gogniat. Machine learning for security: the case of side-channel attack detection at run-time. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems*, pages 485–488. IEEE, 2018.

[107] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. WHISPER: a tool for run-time detection of side-channel attacks. *IEEE Access*, 8:83871–83900, 2020.

[108] Thomas Moscibroda Onur Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *16th USENIX Security Symposium*, 2007.

[109] NIST National Vulnerability Database. CVE-2014-0160 Detail. `https://nvd.nist.gov/vuln/detail/CVE-2014-0160`. Last accessed 29 Jul 2021.

[110] NIST National Vulnerability Database. CVE-2014-3566 Detail. `https://nvd.nist.gov/vuln/detail/CVE-2014-3566`. Last accessed 29 Jul 2021.

[111] Michael Neve and Jean-Pierre Seifert. Advances on access-driven cache attacks on AES. In *International Workshop on Selected Areas in Cryptography*, pages 147–162. Springer, 2006.

[112] The Hacker News. Here's how solarwinds hackers stayed undetected for long enough. `https://thehackernews.com/2021/01/heres-how-solarwinds-hackers-stayed.html`. Last accessed 29 Jul 2021.

[113] NIST. CVSS v3 Equations. `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`. Last accessed 29 Jul 2021.

[114] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2020.

[115] 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 SIGSAC Conference on Computer and Communications Security*, pages 1406–1418, 2015.

[116] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[117] John O'Loughlin and Lee Gillam. Sibling virtual machine co-location confirmation and avoidance tactics for public infrastructure clouds. *The Journal of Supercomputing*, 72(3):961–984, 2016.

[118] D Page. Partitioned cache architecture as a side-channel defence mechanism. 2005.

[119] Mathias Payer. HexPADS: a platform to detect "stealth" attacks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 138–154. Springer, 2016.

[120] Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. Perspectives on the SolarWinds incident. *IEEE Security and Privacy*, 19(2):7–13, 2021.

[121] Colin Percival. Cache missing for fun and profit, 2005.

[122] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-CPU attacks. In *25th USENIX Security Symposium*, pages 565–581, 2016.

[123] Moinuddin K Qureshi. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pages 775–787. IEEE, 2018.

[124] 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*, pages 199–212, 2009.

[125] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):1–34, 2012.

[126] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-leak forwarding: leaking data on meltdown-resistant CPUs (updated and extended version). *arXiv preprint arXiv:1905.05725*, 2019.

[127] Michael Schwarz and Daniel Gruss. How trusted execution environments fuel research on microarchitectural attacks. *IEEE Security and Privacy*, 18(5):18–27, 2020.

[128] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: read arbitrary memory over network. In

European Symposium on Research in Computer Security, pages 279–299. Springer, 2019.

[129] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.

[130] IBM Security. Cost of data breach report 2020, 2020.

[131] Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. A study on microarchitectural covert channel vulnerabilities in infrastructure-as-a-service. In *International Conference on Applied Cryptography and Network Security*, pages 360–377. Springer, 2020.

[132] Benjamin Semal, Konstantinos Markantonakis, Keith Mayes, and Jan Kalbantner. One covert channel to rule them all: a practical approach to data exfiltration in the cloud. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 328–336. IEEE, 2020.

[133] Benjamin Semal, Konstantinos Markantonakis, Raja Naeem Akram, and Jan Kalbantner. Leaky controller: cross-VM memory controller covert channel on multi-core systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, volume 580, page 3. Springer Nature, 2020.

[134] Amazon Web Service. Amazon EC2 dedicated instances. `https://aws.amazon.com/ec2/pricing/dedicated-instances/`. Last accessed 29 Jul 2021.

[135] Amazon Web Service. Monitoring your instances using CloudWatch. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-cloudwatch`. Last accessed 29 Jul 2021.

[136] Carlton Shepherd, Jan Kalbantner, Benjamin Semal, and Konstantinos Markantonakis. A side-channel analysis of sensor multiplexing for covert channels and application fingerprinting on mobile devices. *arXiv preprint arXiv:2110.06363*, 2021.

[137] Carlton Shepherd, Benjamin Semal, and Konstantinos Markantonakis. Investigating black-box function recognition using hardware performance counters. *arXiv preprint arXiv:2204.11639*, 2022.

[138] Archana Singhal, Hema Banati, et al. Fuzzy logic approach for threat prioritization in agile security framework using DREAD model. *arXiv preprint arXiv:1312.6836*, 2013.

[139] Jonathan M Spring, Allen Householder, Eric Hatleback, Art Manion, Madison Oliver, Vijay Sarvapalli, Laurie Tyzenhaus, and Charles Yarbrough. Prioritizing vulnerability response: A stakeholder-specific vulnerability categorization (version 2.0). Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA, 2021.

[140] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural minefields: 4K-aliasing covert channel and multi-tenant detection in IaaS clouds. In *NDSS*, 2018.

[141] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3(3):219–234, 2019.

[142] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[143] Yukiyasu Tsunoo. Crypt-analysis of block ciphers implemented on computers with cache. *Proc. ISITA2002, Oct.*, 2002.

[144] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers

with cache. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 62–76. Springer, 2003.

[145] Johanna Ullrich, Tanja Zseby, Joachim Fabini, and Edgar Weippl. Network-based secret communication in clouds: a survey. *IEEE Communications Surveys and Tutorials*, 19(2):1112–1144, 2017.

[146] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, pages 991–1008, 2018.

[147] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy*, pages 54–72. IEEE, 2020.

[148] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium*, pages 1041–1056, 2017.

[149] Joop Van de Pol, Nigel P Smart, and Yuval Yarom. Just a little bit more. In *Cryptographers' Track at the RSA Conference*, pages 3–21. Springer, 2015.

[150] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium*, pages 913–928, 2015.

[151] Tsvetoslava Vateva-Gurova, Neeraj Suri, and Avi Mendelson. The impact of hypervisor scheduling on compromising virtualized environments. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications;*

Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing, pages 1910–1917. IEEE, 2015.

[152] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 41–46, 2011.

[153] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.

[154] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, pages 225–236. IEEE, 2014.

[155] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference*, pages 1–6, 2016.

[156] Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 142–151. IEEE, 2012.

[157] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *2006 22nd Annual Computer Security Applications Conference*, pages 473–482. IEEE, 2006.

[158] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[159] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on AES in virtualization environments. In *International Conference on Financial Cryptography and Data Security*, pages 314–328. Springer, 2012.

[160] Michael Weiß, Benjamin Weggenmann, Moritz August, and Georg Sigl. On cache timing attacks considering multi-core aspects in virtualized embedded systems. In *International Conference on Trusted Systems*, pages 151–167. Springer, 2014.

[161] Steffen Wendzel, Sebastian Zander, Bernhard Fechner, and Christian Herdin. Pattern-based survey and categorization of network covert channel techniques. *ACM Computing Surveys*, 47(3):1–26, 2015.

[162] Nils Wistoff, Moritz Schneider, Frank K Gürkaynak, Luca Benini, and Gernot Heiser. Prevention of microarchitectural covert channels on an open-source 64-bit RISC-V core. *arXiv preprint arXiv:2005.02193*, 2020.

[163] John C Wray. An analysis of covert timing channels. *Journal of Computer Security*, 1(3-4):219–232, 1992.

[164] 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*, 23(2):603–615, 2014.

[165] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[166] 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 workshop on Cloud computing security workshop*, pages 29–40, 2011.

[167] Zhang Xu, Haining Wang, and Zhenyu Wu. A measurement study on co-residence threat inside the cloud. In *24th USENIX Security Symposium*, pages 929–944, 2015.

[168] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): defending against cache-based side channel attacks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*, pages 347–360. IEEE, 2017.

[169] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy*, pages 888–904. IEEE, 2019.

[170] Yuval Yarom and Naomi Benger. Recovering OpenSSL ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptol. ePrint Arch.*, 2014:140, 2014.

[171] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, pages 719–732, 2014.

[172] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[173] Younis A Younis, Kashif Kifayat, and Abir Hussain. Preventing and detecting cache side-channel attacks in cloud computing. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, pages 1–8, 2017.

[174] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. Truspy: cache side-channel information leakage from the secure world on ARM devices. *IACR Cryptol. ePrint Arch.*, 2016:980, 2016.

[175] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: a real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 118–140. Springer, 2016.

[176] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Memory DoS attacks in multi-tenant clouds: severity and mitigation. *arXiv preprint arXiv:1603.03404*, 2016.

[177] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on ARM and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 858–870, 2016.

[178] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K Reiter. Homealone: co-residency detection in the cloud via side-channel analysis. In *2011 IEEE symposium on security and privacy*, pages 313–328. IEEE, 2011.

[179] 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 2012 ACM conference on Computer and communications security*, pages 305–316, 2012.

[180] 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*, pages 990–1003, 2014.

[181] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 871–882, 2016.