Télécom SudParis

Internship Report

Implementation of a Moving Target Defense Framework to Defend Against Isolation Violations in IaaS Infrastructures

Author:
Thibaut Sautereau

Supervisor:
Mohammad-Mahdi Bazm

University Supervisor:
Hervé Debar

Orange Labs and Services

February 1, 2017 - July 31, 2017

Abstract

Cloud computing is based on the sharing of physical resources among several virtual machines (VMs) through a virtualization layer providing software isolation. Research on cache-based side-channel attacks has shown that the cache architecture in processors leaks information, allowing a malicious VM to obtain sensitive data such as cryptographic keys from another VM co-located on the same physical machine. The objective of this internship is to implement a detection module based on Gaussian anomaly detection that will be integrated into a Moving Target Defense (MTD) framework. An MTD approach aims to move a suspicious VM to other physical machines in order to interrupt attacks relying on co-residency (such as cache-based side-channel ones). In this report, we provide the state of the art of cache-based side-channel attacks and existing countermeasures. We propose a detection module that takes advantage of Intel Cache Monitoring Technology (CMT) and Hardware Performance Counters (HPCs) to detect cache-based side-channel attacks. Finally, we evaluate the efficiency of the module and discuss how it can be further improved. Experimental results show that our approach can detect cache-based side-channel attacks with a low performance overhead (≈ 2 %).

Keywords: cache-based side-channel attacks, cloud computing, Cache Monitoring Technology, Moving Target Defense

Résumé

L'informatique dans les nuages repose sur le partage de ressources physiques entre plusieurs machines virtuelles (VM) grâce à une couche de virtualisation fournissant une isolation logicielle. Les recherches sur les attaques par canaux auxiliaires visant les caches CPU ont montré que des informations sensibles telles que des clés cryptographiques pouvaient fuiter des caches CPU d'une VM et ainsi être récupérées par une autre VM malveillante située sur la même machine physique. L'objectif de ce stage est d'implémenter un module de détection basé sur la détection d'anomalie avec la loi Gaussienne, qui sera ensuite intégré à un système de *Moving Target Defense* (MTD). Une approche MTD consiste à déplacer une VM suspecte vers une autre machine physique afin d'interrompre les attaques basées sur la co-localisation (telles que les attaques par canaux auxiliaires). Dans ce rapport, nous fournissons l'état de l'art des attaques par canaux auxiliaires ciblant les caches CPU ainsi que des contremesures existantes. Nous proposons un module de détection tirant profit d'une technologie d'Intel nommée *Cache Monitoring Technology* (CMT) ainsi que des *Hardware Performance Counters* (HPC) afin de détecter des attaques par canaux auxiliaires ciblant les caches CPU. Finalement, nous évaluons l'efficacité de notre module et discutons des améliorations qui pourraient y être apportées. Les résultats de nos expériences montrent que notre approche peut détecter des attaques par canaux auxiliaires ciblant les caches CPU avec un faible surcoût au niveau des performances (≈ 2 %).

Mots-clés : attaques par canaux auxiliaires sur caches CPU, informatique dans les nuages, *Intel Cache Monitoring Technology*, *Moving Target Defense*

Acknowledgements

The internship opportunity I had with Orange Labs and Services was a great chance for learning and professional development. First and foremost, I would like to express my deepest gratitude and special thanks to my supervisor Mohammad-Mahdi BAZM for his helpful comments, remarks, support and commitment during my internship. Furthermore, I would like to thank my team manager Sébastien ALLARD for his help and for allowing me to feel great in the team since the first day of my internship. I am also very grateful to Marc LACOSTE for his precious insight during various discussions and meetings on the technical parts and the correct way of doing research on my subject.

I would also like to thank all members of the SDS and NCP teams for all those nice moments and interesting discussions. I consider myself very lucky to have had the opportunity to work in such a good atmosphere.

Finally, I would like to express my gratitude to my university supervisor Hervé DEBAR for accepting to be my supervisor, for his time and for giving useful directions during the internship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

My internship took place in a security research team of Orange Labs and Services, on Orange's new campus *Orange Gardens* in the city of Châtillon (France, 92320).

1.1 Presentation of the Company

Orange is a French multinational telecommunications corporation. It has more than 260 million customers worldwide[1] and this number keeps increasing. In 2013, the company was the leader or the second operator in 75 % of the European countries it was established in, and in 83 % of African and Middle East countries. In 2016, its turnover was around 41 billion euros[2].

Orange is positioned on a wide range of services related to telecommunications through its various subsidiaries, mobile phone operator (201 million customers in 28 countries) and broadband internet service provider (18.1 million customers, including 3.3 million connected trough fiber connection in Europe). Through Orange Business Services, the group also offers a variety of business services, from cloud infrastructure hosting (SaaS, PaaS and Iaas) to end-to-end connectivity across company sites.

1.1.1 Research Activities

Orange's research and development activities are carried out by three entities: Orange Labs Services, Orange Labs Networks and Orange Labs Research. These entities have research centers in twelve countries and are accountable for more than 8,500 patents. These innovations are possible thanks to the work of more than 5,000 employees dedicated to innovation and to 726 million euros invested in 2015.

Orange Labs Research is responsible for research management and directs the group's research activities. **Orange Labs Networks** is in charge of standardization and tendering processes and is focused on networks and communications services. Finally, **Orange Labs Services**[3] concentrates on anticipation and delivery of other services.

1.1.2 Teams

I was part of the team identified by the following: IMT/OLS/ARSEC/SEC/SDS.

IMT stands for *Innovation, Marketing et Technologies* and is the division of Orange which is responsible for all the innovation activities of the group.

OLS is one of the many directions of IMT, stands for Orange Labs Services and was already mentioned.

[1]http://www.orange-business.com/fr/les-chiffres-cles

[2]http://www.lemondeinformatique.fr/actualites/lire-annuels-orange-2016-le-ca-entreprise-tire-par-le-cloud html

[3]Orange Labs Services was named Orange Labs Products and Services until May 2017

ARSEC stands for Architectures, Enablers and Security. Its main objectives are to define target architectures and technical policies for services and information systems, by integrating network transformations, contributing to technical strategies of the various OLS directions and bringing support and consulting.

SEC is the security department of ARSEC. Its 103 employees are based in Châtillon, Caen, Lannion, Rennes and Blagnac. SEC carries out research and development activities on the group's IoT, devices and boxes products, on cryptography and anonymization, on normalization, on Cloud security, on network security and resilience (especially for 5G and SDN/NFV) architectures), etc.

Finally, I was in the **SDS** (System and Device Security) team based in Châtillon. The team shares an open space with another team of the SEC department and is currently composed of 27 people, more than half of them being PhD students, apprentices or interns. Some of them audit Orange's IoT and mobile devices as well as Liveboxes (ADSL wireless routers). Other focus on normalization (IoT, Mobile Networks, eSIM, etc.) or on research and development activities regarding cryptography, network and Cloud security, virtualization and multiclouds, side-channel attacks, etc. The three other teams of the SEC department are named Network Cybersecurity and Privacy (based in Rennes and Châtillon), Security Enhanced Applied (based in Lannion and Blagnac) and Security Privacy Innovation (based in Caen).

Teams based in Châtillon work at *Orange Gardens*, the group's new Eco Campus inaugurated in June 2016. It is devoted to innovation and mainly hosts teams from the IMT division of Orange.

1.2 Presentation of the Internship

1.2.1 Internship Goals

Side-channel attacks constitute a very active field of research, especially processor cache-based ones. Side-channel attacks are a particular class of attacks aiming at gathering information that should not be accessible otherwise. Such attacks usually target cryptographic algorithms by exploiting flaws in their implementation, allowing for the recovery of cryptographic keys between processes (potentially belonging to different users) on the same machine. All of that is achieved using legitimate operations, meaning any action that a normal user should be able to accomplish on a normal system. Side-channel attacks are typically based on timing information, power consumption or electromagnetic leaks. *Cache-based side-channel attacks* are a particular class of side-channel attacks targeting CPU caches. They are based on the ability of an adversary to monitor cache accesses made by the victim during the execution of an application (e.g. cryptographic ones) in virtualized environments such as clouds.

Cloud computing is becoming increasingly popular with the dynamic and elastic allocation of resources and services. Fundamental to the economy of clouds is high resource utilization by sharing physical resources among several instances. Virtualization is the key enabling technology that allows virtualized instances, e.g. virtual machines (VMs) to run stand-alone operating systems (OSs) through a complex software layer called *VM Monitor* (VMM) or *hypervisor*.

Due to the nature of cloud computing, the underlying virtualization layer is expected to guarantee strong **isolation** between VMs. But it always remains a sort of illusion and in reality, the virtual resources map to shared physical resources, creating a leaky channel between VMs co-located on the same physical machine.

Consequently, several efforts [36][50][23] have demonstrated the possibility of information leakage *via* **co-residency** in virtualized environments such as cloud infrastructures. This vulnerability allows a malicious VM to exploit a leaky channel to recover sensitive information such as cryptographic keys from a victim VM co-located on the same physical machine. In particular, cache memories (especially the last-level cache that is shared among all cores) in processors are targeted and exploited as an attack vector.

Several different methods were proposed over the past few years to detect [8][48][51] such attacks or mitigate them in static and dynamic ways. Some of them will be presented and discussed in chapter 2. The first goal of this internship is to try to leverage a very recent and promising approach called **Moving Target Defense** (MTD) to mitigate cache-based side-channel attacks in the cloud. The main idea of MTD is to move the attack surface (instead of trying to reduce it as other approaches propose) by reconfiguring security mechanisms. For instance, a suspicious VM can be migrated to other physical machines in the cloud infrastructure in order to interrupt a potential side-channel attack taking place between two VMs.

However, any MTD approach needs to be equipped with a complementary approach to detect potential malicious VMs. That was the second goal of this internship and it eventually became the one I predominantly focused on: experimenting with a new Intel technology called **Cache Monitoring Technology** (CMT) to see if it could be used to improve existing detection techniques.

1.2.2 Contribution

I started by learning the necessary background on CPU caches and side-channel attacks targeting them, as well as related detection and mitigation methods. This allowed me to assess the state of the art of this area of research and put together a little **survey**.

Then, I spent most of my time searching and exploring potential ways of leveraging CMT as a way to detect cache-based side-channel attacks. This led me to the development of a **detector prototype** written in C that takes advantage of CMT as well as Hardware Performance Counters, which will be discussed further in this report. This detection module uses `libvirt` in order to be more generic and more easily adaptable to multiple hypervisors[4]. During the conception of the detection module, I also studied several anomaly detection methods leveraging machine learning techniques such as the Gaussian distribution, which I focused on. The reasons behind this choice will be explained later. The detector was tested through various experiments that will be detailed. Its ability to detect cache-based side-channel attacks as well as its performance will be presented.

In the remaining of this report, Chapter 2 provides a state of the art on cache-based side-channel attacks, detection and mitigation approaches. In Chapter 3, we present Intel Cache Monitoring Technology and describe our prototype. Then, Chapter 4 presents our experimental results. Finally, Chapter 5 sums up our work and concludes on this internship.

[4]The detection module was initially designed for QEMU-KVM.

Chapter 2

State of the Art of Cache-based Side-channel Attacks, Detection and Mitigations

In this chapter, we give some background information on the structure of caches in processors. Then, we provide a state of the art on cache-based side-channel attacks, detection and mitigation approaches. Finally, we underline the importance of such attacks for virtualized environments such as clouds.

2.1 Background on CPU Caches

Cache memory is vital for modern processors. It is extremely important for performances because it hides the memory accesses latency to the slow physical memory (DRAM[1]) by buffering frequently used data in a small and fast memory (SRAM[2]). Due to locality (spatial or temporal) of reference, instructions or data recently used by the processor are more likely to be reused shortly: storing and then fetching these values from the CPU cache saves time and reduces the pressure on the main memory. Indeed, modern processors are a lot faster than current main memory and the mismatch causes many CPU cycles to be lost waiting for instructions and data to be fetched from or written to DRAM.

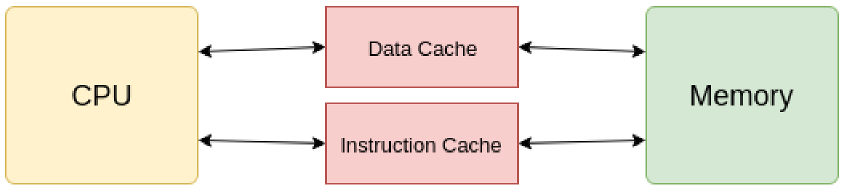


FIGURE 2.1: Harvard architecture

For instance, the Harvard architecture has two separated buses for instructions and data [39]. This increases performances because these resulting caches can simultaneously be accessed through both buses.

However, as such high-speed memory is expensive, the memory hierarchy is organized into multiple levels of cache. Low-level caches are smaller (KiB), faster and closer to the CPU cores than high-level caches which are larger (KiB) in capacity and closer to the main memory. As shown in Figure 2.2, each core typically has two private top-level (level-1 or L1) caches, one for data and another for instructions. In

[1]Dynamic Random-Access Memory
[2]Static Random-Access Memory

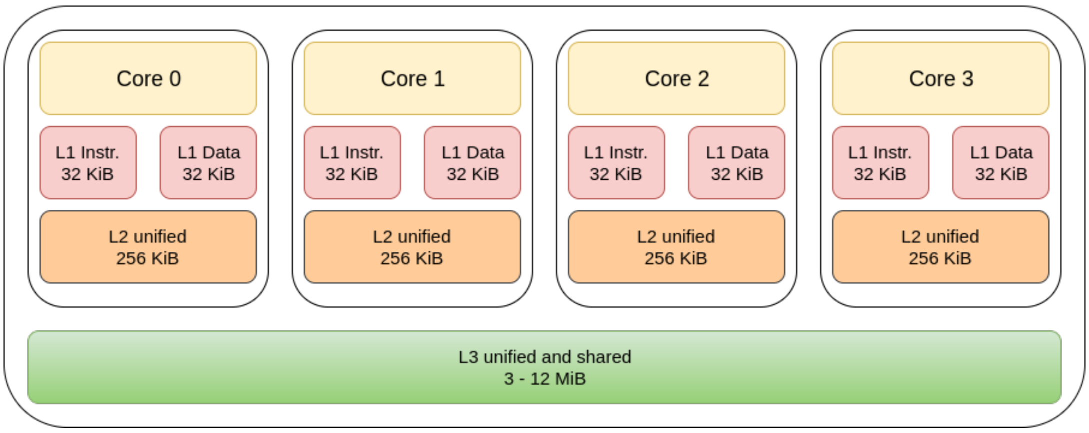


FIGURE 2.2: Intel Skylake cache architecture (simplified)

Intel Core and Xeon families, the L1 cache size is 32 KiB with a 4-cycles access latency. Each core also has a unified (i.e. containing both data and instructions) level-2 (L2) cache whose size is usually 256 KiB with a 12-cycles access latency. Finally, the level-3 (L3) cache, also named *last-level cache*[3] (LLC), is **shared** among all cores of the same CPU (or socket). LLC has several megabytes (MiB) as size and a 40-cycles access latency.

Apart from the aforementioned caches, there are also the *Translation Lookaside Buffers* (TLBs). They also are caches but they store the recent translations of virtual memory to physical memory and thus can be called address-translation caches. In modern Intel microarchitectures, there are two[4] TLBs: one for instructions (ITLB) and one for data (DTLB). Moreover, although TLBs could be located between any levels of cache memory, they usually sit between the CPU core and the L1 caches.

Whenever the CPU requests data from a given cache, if data is present in this cache, a *cache hit* occurs. Otherwise, data has to be fetched from a higher-level cache or from the main memory if it is not present in the LLC: this is a *cache miss*. In the TLB case, a miss causes a page walk. In any case, this is time consuming and obviously not good for performances.

In the following, we focus on Intel CPUs. Indeed, the vast majority of the literature on cache-based side-channel attacks focuses on Intel processors because their cache architecture is **inclusive**. This means that an upper-level cache contains copies of all of the data stored in the lower-level caches. That is, data present in the L1 cache is also present in the L2 cache as well as well as in the LLC. As we explain further, the inclusiveness of CPU caches plays an important role in the implementation of cache-based side-channel attacks. Because AMD processors use exclusive caches, performing cache-based side-channel attacks on such processors is more difficult (for instance, a given cache line can reside in one of the L1 and L2 caches, but never in both).

CPU caches are organized as *sets* of *cache lines*, as shown in Figure 2.3. A cache line is the smallest unit in the cache organization. Nowadays, CPU caches are generally **set-associative**. It means that each memory location can be cached in a restricted number of cache entries. Although there are different cache allocation strategies, we only focus on classical Intel CPUs. A *N-way* set-associative cache has N cache lines in each set. These lines are stored in cache banks called *ways*. A 1-way

[3]level-4 caches are still quite rare
[4]On most recent Intel Skylake processors, there is also a unified L2 TLB (STLB)

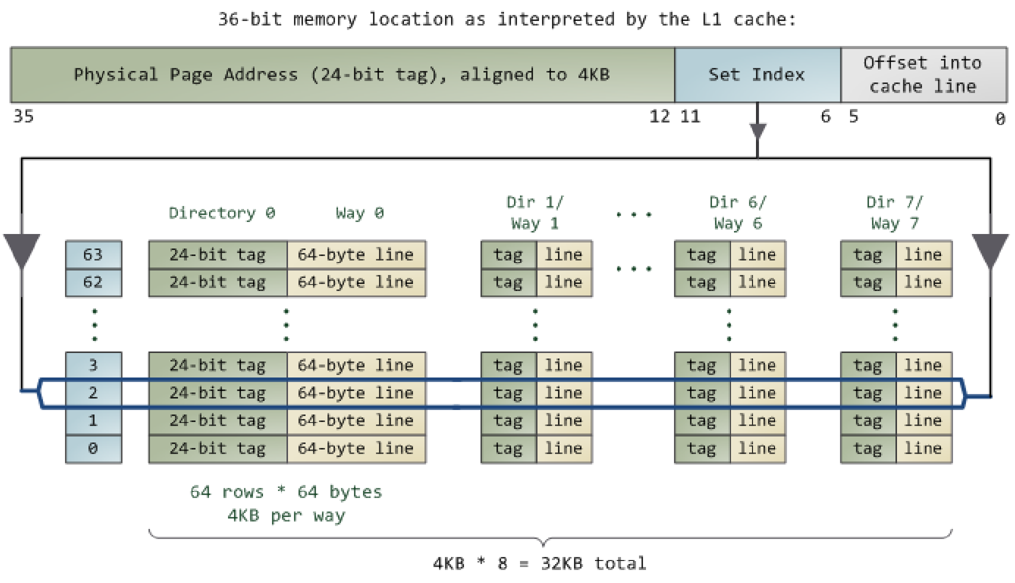


FIGURE 2.3: L1 cache, 32 KiB, 8-way associative, 64-byte cache lines [10]

set-associative cache is also called a *direct-mapped cache* (each location in the main memory can only be mapped to one entry in the cache). In a *fully-associative cache*, each location in the main memory can be mapped to any entry in the cache. The type of associativity directly impacts performances. As the degree of associativity rises, more cache entries need to be checked when looking for a cache line. However, as the degree of associativity decreases, more cache misses can be expected. Low latency and low power consumption are very important factors in the improvement of cache efficiency. For instance, Intel Skylake CPUs have 8-way set-associative L1 caches, 4-way set-associative L2 caches and a 16 way set-associative L3 cache.

The $log_2 B$ (B is the size of a cache line) least significant bits (LSBs) of a virtual memory address, called *line offset*, are used to locate a byte in the cache line. Then, following this logic in the case of a 64-sets cache, bits 6 to 11 determine the set to be accessed. Then, the correct cache line in the set still needs to be found. This happens using the *tag*, which is stored in each cache bank together with the cache line. In traditional L1 caches, the tag is the remaining most significant bits (MSBs) of the virtual memory address.

We detailed *indexing* and *tagging* through practical examples. However, CPU caches can be virtually or physically indexed. It means respectively that the virtual or physical memory address is used to derive the index. The same derivation process is valid for tags. As we will see later, this is important for performances and it is also significant for attack purposes. For instance, L1 caches usually are virtually indexed but physically tagged. This allows parallelizing cache line lookups: the process of finding the correct set can be started without waiting for the MMU to translate the virtual memory address into a physical one. However, using a physical tag is important to uniquely identify a physical memory page. To conclude this topic, virtual indexing works well as long as a cache way is not bigger than a MMU page. Otherwise, a given physical memory location is not guaranteed to be associated with the same set, depending on its virtual mapping. Thus, L2 and L3 caches need to be physically tagged and physically indexed.

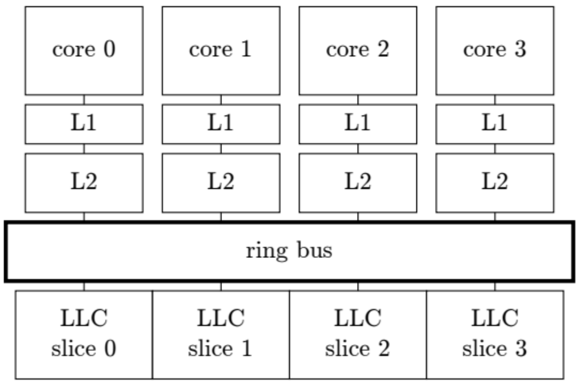


FIGURE 2.4: Simplified cache architecture of a quad-core Intel processor (since Sandy Bridge microarchitecture) [29]

Obviously, main memory is much larger than any cache and in the case of a N-way set-associative cache, more than N memory lines map to the same cache set. This leads to conflict misses and consequently, a heuristic needs to be used to choose an entry for evicting from the cache in order to make room for the new one. This is called a **cache-replacement policy**. The main challenge of such policies is how to efficiently replace cache lines (e.g. keep the ones that are likely to be reused in the near future). A popular policy is the Least-Recently Used (LRU) policy, which replaces the least recently used cache entry. This requires keeping information on each cache line in order to know their "age". It can become costly if the cache has many ways. For instance, Intel LLC caches implement the pseudo-LRU policy, which almost always discards *one of* the least recently used cache entries. ARM LLC caches usually implement a pseudo-random replacement policy, where a random cache entry will be selected and evicted based on a pseudo-random number generator. Cache-replacement policies are very important since they strongly impact the implementation of cache-based side-channel attacks.

In Intel processors (since the Sandy Bridge family), the LLC is divided into a certain number of equal *slices* according to the number of CPU cores. As shown in Figure 2.4, these slices are interconnected by a *ring bus* and contain sets like the other cache levels. Physical addresses are mapped across the slices using an undocumented *hash function* with cache line granularity. As a consequence, consecutive cache lines will be mapped to different LLC slices. The hash function was designed to distribute the traffic approximately evenly among the slices and reduce congestion, no matter what the access pattern [22]. This is a *complex addressing* scheme (in contrast to direct addressing, which is still used for the sets). All address bits are generally used to determine the cache slice, excluding the least-significant bits that determine the offset in a cache line (cf. Figure 2.5). The use of multiple slices is intended to increase bandwidth, not to reduce latency. Unloaded latency is typically slightly higher with a multi-slice cache, since there is no practical way to prevent most of accesses from being to remote LLC slices. However, this is widely counterbalanced by the increased throughput of the multi-slice LLC.

In modern systems, hardware implements shared memory allowing each CPU core to read and write in a single shared address space. Consequently, precautions must be taken to guarantee that multiple cached copies of data present in different cores are up-to-date. It is the role of *cache coherence protocols*. They perform at the cache line granularity and must ensure write propagation and transaction serialization. We

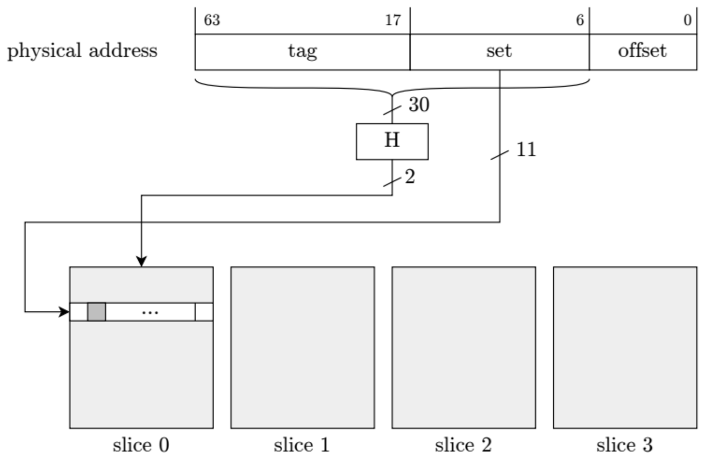


FIGURE 2.5: Complex addressing scheme in the LLC with 64B cache lines, 4 slices and 2048 sets per slice [29]

are interested in the following actions on cache lines:

- **Invalidate**: means simply marking a cache line as invalid, i.e. accessing it will trigger a cache miss.
- **Clean**: this means writing back the content of the cache line to main memory, but only if it is *dirty*[5].

The term *evict* is also used when a cache line is replaced with another one by the cache replacement policy. *Flushing* generally means cleaning and then invalidating a cache line.

There are two main cache coherence mechanisms: *snooping* and *directory-based*. The first one monitors the bus, is faster but does not scale well. The second is typically used in NUMA or large multi-core systems. The snooping coherence mechanism is composed of two types of coherence protocols: write invalidate (MSI, MESI, MOSI, MOESI, etc.) and write update. In any case, this is handled by the hardware itself.

However, the software layer may need to invalidate specific cache lines. In the Intel x86 architecture, the `clflush` instruction can be used at all privilege levels. It invalidates the cache line that contains the linear address specified with the source operand from all levels of the cache hierarchy. The invalidation is broadcasted throughout the cache coherence domain. If, at any level of the cache hierarchy, the line is inconsistent with memory (dirty), it is then written to memory before invalidation. As we see further in this report, this particular instruction is widely used in many cache-based side-channel attacks especially because it is accessible to userspace. For instance, it is not the case in ARM systems: two different privileged instructions are used for invalidating and cleaning cache lines.

2.2 Cache-based Side-channel Attacks

Cache-based side-channel attacks exploit weaknesses in the microarchitectures of processors to provoke information leakage. There are several types of such attacks but all

[5]In memory management, *dirty* means data that has changed but has not been written back to its primary location, and thus is out-of-sync.

of them are based on the same principle: since data that is present in the cache can be accessed much faster than data that needs to be loaded from the main memory, it is possible to infer whether a specific portion of data resides in the cache. If so, it implies that the data has been recently accessed. A great advantage of cache-based side-channel attacks is that they do not require physical access to target computers or direct access to the memory space of victim processes: they are conducted through legitimate software operations.

Cache-based side-channel attacks can be classified into three categories:

- **time-driven**: The basic idea of such attacks is to gather timing information on many encryptions and then to perform statistical correlations in order to recover the used secret key. They typically require many measurement samples.
- **access-driven**: These attacks aim at determining which cache lines or cache sets have been accessed during the encryption. Therefore, they require knowledge of the memory layout of the victim program as well as information about the cache architecture. To derive a profile of cache activities, the attacker manipulates the cache by evicting cache lines of the victim process.
- **trace-driven**: Such attacks need a detailed cache profile based on the information of every single memory access. An offline phase then permits to infer the secret data that was used by the victim process.

All cache-based side-channel attacks described below need to measure time with the highest possible resolution. On x86, the unprivileged `rdtsc` instruction is used for that purpose and can obtain a sub-nanosecond timestamp.

2.2.1 Evict+Time

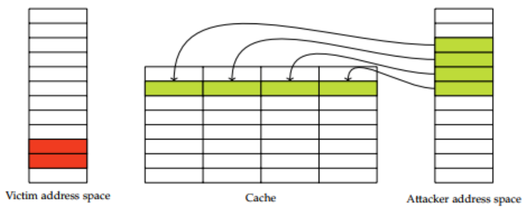
The first cache-based side-channel attack was called *Evict+Time* and was formalized by Osvik et al. [33] in 2005. The basic approach is to determine which cache set is used during the victim's computations. In other words, the first step is to measure the execution time of the victim program. Then, a specific cache set is evicted and the execution time of the program is measured again. Based on the timing difference between the two measurements, an adversary may deduce how much the specific cache set is used while the victim program is running.

This technique was mainly used to attack AES implementations in OpenSSL [13]. The main drawback of this attack is that it is absolutely not fine-grained and is rather a post-mortem technique. However, it can be a good alternative to the other methods when facing caches with a (pseudo-)random replacement policy (employed by many ARM processors for the LLC). For instance, Spreitzer et al. [40] managed to recover AES key bits by tracing memory-access patterns of disaligned T-tables on ARM Cortex-A series processors.

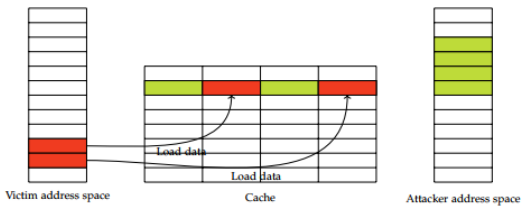
2.2.2 Prime+Probe

Another cache-based side-channel attack that was described by Osvik et al. [33] is *Prime+Probe*. It allows an attacker to determine which cache sets are used while the victim application is executing. Basically, the attacker occupies specific cache sets with its own data, schedule the victim program, and access again these cache sets in order to determine which ones are still occupied by its data.

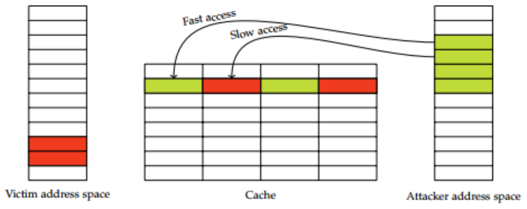
The *Prime+Probe* attack is illustrated in Figure 2.6. It represents a 4-way cache with 8 sets. The attacker performs the attack on the 2^{nd} cache set. In the *prime*



(a) `Prime`: Occupy the cache set



(b) Schedule the victim's program that accesses data in the same cache set.



(c) `Probe`: Determine cache set occupation.

FIGURE 2.6: Illustration of the *Prime+Probe* attack (4-way set-associative cache) [26]

phase (`a`), he fills this set with his data by accessing congruent memory addresses[6] that all map to this cache set.

In the second phase (`b`), the victim's program runs and thus fills cache sets with application memory lines. If the program uses memory addresses that map to the 2nd cache set, it will consequently evict cache lines belonging to the attacker.

Finally, in the *probe* phrase (`c`), the attacker reaccesses the same memory addresses that he used in the first phase. Depending on the time it takes to reaccess an address, he can infer whether the corresponding data was pulled from the cache or had to be fetched from the upper-level cache (or from the main memory if the attack was targeting the LLC). The time period between steps `b` and `c` is very important and must be precisely chosen by the attacker. Otherwise, he is unable to capture the victim's cache activities.

This technique usually targets the L1 or TLB caches (or even branch prediction caches) as they are less noisy and easier to probe because of their small sizes (KiB). However, such caches are core-private and the attack thus needs to exploit either hyper-threading[7] or time multiplexing of the core. In both cases, the attacker nearly always needs to exploit scheduler vulnerabilities [33][52][1].

2.2.3 Flush+Reload

The *Flush+Reload* attack derives from the *Prime+Probe* attack. It relies on the basic idea of using the `clflush` instruction to run an access-driven cache attack, as proposed by Gullasch et al. [20]. The *prime* phase is replaced with a *flush* phase, that leverages the `clflush` instruction to flush memory lines of the victim process from all the cache hierarchy. Then, the attacker waits for a precise time period (defined by the attacker) while the victim process reaccesses memory lines. Finally, the attacker reloads the memory lines that were evicted from the cache to determine which lines were accessed by the victim process during the second phase (cf. Figure 2.6).

[6]Virtual or physical depending on the indexing of the cache
[7]Intel's proprietary Simultaneous Multi-Threading (SMT) implementation

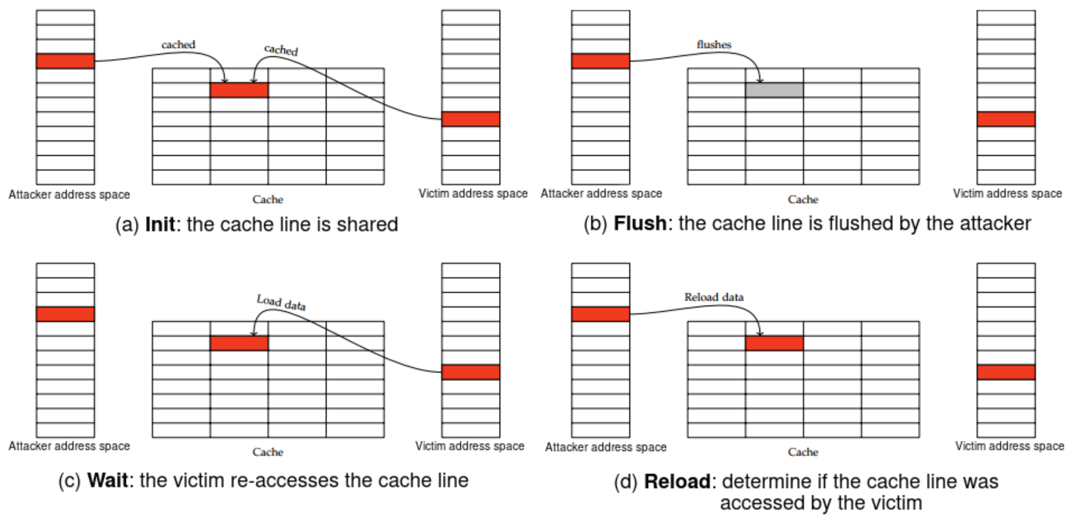


FIGURE 2.7: Illustration of the *Flush+Reload* attack (4-way set-associative cache)

Yarom and Falkner [50] extended this idea by taking advantage of the cache architecture in Intel processors to target the LLC: since caches are inclusive and the LLC is shared across cores, the attacker program can carry out an attack without the need to share the execution core with the victim. Their attack also leverages *page sharing* [7]. Indeed, to perform the attack without any knowledge of the victim's virtual memory mappings, the attacker must share the same virtual address space as the victim. This is obtained with *page deduplication*[8] [3], an extended form of page sharing that scans the active memory to identify and merge pages with different ownerships but with identical content. Typically, an attacker would map a binary (e.g. a shared object or an executable) which is used by the victim, using a system call such as `mmap`, and then expect the corresponding physical memory pages to be merged with the victim memory pages.

The main advantages of the *Flush+Reload* technique are: it efficiently targets the LLC (i.e. there is no need to exploit a scheduler weakness) ; it is fine-grained (especially for a LLC attack, allowing attacks against cryptographic algorithms as well as keylogging) thus it only requires a few number of samples to retrieve sensitive information such as a secret key ; it does not need to bypass diversification techniques such as ASLR[9] [50] because of page deduplication. However, it works only on inclusive caches and when page deduplication is enabled. This is why cloud providers prefer to disable page deduplication for security reasons to guarantee isolation between different users, although this really decreases memory efficiency.

2.2.4 Evict+Reload

The *Evict+Reload* attack was introduced by Gruss et al. [18]. It replaces the `clflush` instruction in the *Flush+Reload* technique with cache line evictions. This attack has no practical use on x86 CPUs because the `clflush` instruction is unprivileged. However and as we mentioned earlier, that is not the case for ARM processors.

[8]Page deduplication is implemented as Kernel Same-page Merging (KSM) in Linux and as Transparent Page Sharing (TPS) in VMWare.

[9]Address Space Layout Randomization

The efficiency of this technique highly depends on the cache replacement policy. Usually, the LLC in ARM microarchitectures uses a pseudo-random policy. Therefore, the eviction rate can be very slow and result in a high false positives rate [26].

2.2.5 Flush+Flush

Both *Flush+Reload* and *Prime+Probe* attacks provoke numerous cache references and cache misses [17], which can be monitored and detected as we will see further in this report. Gruss et al. presented *Flush+Flush* [17] as a new technique to implement a cache-based side-channel attack. It is basically the same as *Flush+Reload* except that it relies on the fact that the execution time of the `clflush` instruction varies depending on whether the data is cached or not: if the source operand is a memory address corresponding to cached data, the instruction will take more cycles because it has to trigger eviction for all local caches. This approach is faster, does not trigger prefetches (thus allows monitoring multiple addresses within a 4 KiB memory range) and does not cause cache misses for the attacker, which makes it stealthier.

However, Anders Fogh [12] showed that *Flush+Flush* can still be detected because it always uses the same structure, i.e. `clflush` instruction bracketed by `rdtsc` instructions in short sequences. Moreover, this attack still causes cache misses for the victim and it inherently has a lower accuracy than *Flush+Reload*. Indeed, both the timing difference between a hit and a miss and the average access time are lower than in the *Flush+Reload* case. Nevertheless, this can be counterbalanced by monitoring several nearby memory addresses. *Flush+Flush* requires page sharing as *Flush+Reload*.

2.2.6 Cross-VM Side-channel Attacks in Clouds

As any emerging technology, cloud services have encountered their security challenges. In cloud computing, VMs are co-located on the same physical machine through a software abstraction layer (i.e. virtualization layer) that shares physical resources (e.g. CPU and RAM) among them. The virtualization layer is expected to provide isolation among VMs. However, Ristenpart et al. [36] experimented with cross-VM information leakage techniques on Amazon EC2 and were the first to successfully implement side-channel attacks inside VMs. They used network probing to map the EC2 service and implemented network-based co-residency checks (which were verified using a hard-disk-based cross-VM covert channel). They also exploited the VM placement policy to co-localize victim and attacker VMs on the same physical machine in EC2. They adapted existing cache-based side-channel attacks to cloud computing. Although coase-grained, these attacks allowed them to obtain a covert channel, to implement load-based co-residency detection and to estimate traffic rates. It has created a new research interest on novel techniques of implementing side-channel attacks and adapting them to the cloud environment since 2009.

Zhang et al. [52] proposed a cross-VM attack using the *Prime+Probe* technique that exploits the square-and-multiply implementation in `GnuPG 1.4.13` through the L1 instruction cache. Their attack relies on exploiting a vulnerability in the Xen scheduler to obtain co-residency on a single core. It requires six hours of constant decryptions for collecting enough data to break a secret key.

On a virtualized host, VMs usually have their virtual CPUs (vCPUs) pinned to CPU cores (pCPUs). It means that VMs do not share any core-private caches such as TLBs, L1 and L2 caches. However, they share the LLC. This is true for all VMs whose vCPUs are pinned to physical cores of the same CPU socket (cf. Figure 2.8). The

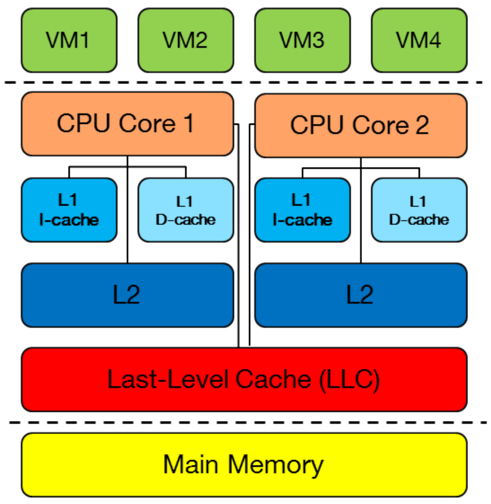


FIGURE 2.8: Shared LLC between VMs co-located on the same physical machine

LLC is thus a very interesting target for cache-based side-channel attacks. However, it is larger and slower than L1 and L2 caches because of its higher degree of associativity and its longer access latency.

Yarom and Falkner's approach [50] can be efficiently used to target the LLC with a high resolution, but it requires the hypervisor to enable page deduplication.

Irazoqui et al. [23] presented an efficient cross-core *Prime+Probe* attack that does not take advantage of page deduplication features. It is non-intrusive because only 4 cache sets in the LLC need to be monitored to recover a full AES encryption key. Their attack has a higher resolution because the access-time gap is more important between the LLC and the main memory than it is between the L1 and L2 caches. They also implement a "re-prime phase" to ensure that the attacker's data only resides in the LLC in order to have a lower bound for the access time, thus avoiding false positives or false negatives. However, they exploit the architecture of the LLC in Intel Nehalem processors, which do not use complex addressing for slices. Therefore, their attack cannot succeed with later Intel microarchitectures.

In a similar work, Liu et al. [28] proposed an asynchronous and fine-grained *Prime+Probe* attack on the LLC that does not require sharing cores or memory between the attacker and the victim. Furthermore, their attack does not exploit hypervisor weaknesses and runs on typical server platforms. Additionally, the attack overcomes the difficulties induced by the undocumented LLC hashing schemes in Intel processors since the Sandy Bridge microarchitecture, without reverse-engineering it. This is achieved with an algorithm used by the attacker to build eviction sets and probe exactly one cache set at a time to identify (*via* temporal access patterns) the victim's security-critical accesses. The attacker hence probes the whole LLC only once. Moreover, their attack also leverages large pages as a workaround for hidden mappings. The attack can recover secret-dependent execution paths and also secret-dependent data access patterns. They target the GnuPG *advanced sliding window* technique for modular exponentiation (it was impossible using Yarom and Falkner's *Flush+Reload* technique). However, their attack requires many executions of the victim process to locate the cache set of the sensitive data and was not tested in a real and noisy environment.

Although cache slicing in the Intel LLC makes cache-based side-channel attacks more complex to perform, it may be considered an advantage to increase the resolution of such attacks by reducing the number of sets to probe (for instance, 25 % in the case of 4 slices) [28].

2.3 Detection and Mitigation of Cache-based Side-channel Attacks

From a high-level point of view, there are different approaches for mitigating cache-based side-channel attacks, which can require prior detection:

- **application-based approaches:** They are based on programming methods (e.g. constant-time programming [15] to write robust algorithms that are not threatened by timing attacks) and compilers (e.g. code transformation [9]).
- **system-based approaches:** Such approaches can be guest OS-level countermeasures (time-padding, cache partitioning, adding noise, etc.) or hypervisor-level countermeasures (time-scheduler based, static/dynamic page coloring, cache lines locking, etc.).
- **hardware-based approaches:** They consist of random permutations, static/dynamic cache partitioning, caches lines locking, etc.

System-based and hardware-based approaches mainly focus on avoiding resource contention and some of them will be detailed further in this report.

Gruss et al. [17] pointed out that an efficient mitigation for *Flush+Flush* would be to make `clflush` a constant-time instruction. Another approach could also be to use informing loads (i.e. to raise a user-level exception when a special load instruction misses in the cache) or set the `CR4.TSD` flag to make `rdtsc` privileged and trap on it: that would introduce a certain delay, hopefully causing too much noise and consequently preventing the attack from succeeding. Moreover, since 2008, `seccomp` can be used on Linux to enable the `rdtsc` instruction only in privileged mode.

Alam and Sethi [2] present an approach to generalize parameters for covert channel detection, based on machine learning (Kolmogorov-Smirnov test).

Wu et al. propose C^{2}Detector, a covert channel detection framework in cloud computing [48]. It represents a cache line as an automaton and is based on a two-phases detection: the first phase uses a Markovian detection algorithm with a pessimistic threshold in order to get rid of false negatives ; a second detector based on a Bayesian algorithm is then used to eliminate false positives. The resulting detector can detect all the three types of covert channel attacks defined in the threat scenarios, with good performances. It is also extensible through plugins.

Wang et al. [41] proposed new cache designs (PLcache and RPcache) to prevent information leakage. Although it is effective, the main drawback of such hardware-based defenses if that they lack flexibility to adapt to future attacks.

Kong et al. [25] tackle this issue by developing integrated hardware-software mitigation approaches. First of all, they point out that existing software countermeasures (*access-all* techniques against access-driven attacks, *random permutations* against access-driven attacks, *small tables* against access/time-driven attacks, *preloading* against access or time-driven attacks and combinations of these techniques) cause substantial overhead. They also explain that existing hardware countermeasures such as partitioned cache or Partition-Locked cache (PLcache), and Random-Permutation cache (RPcache), are both not secure. Neither is preloading. As a consequence, they propose to add preloading to protect the PLcache and informing loads to secure the RPcache (as well as regular caches), thus creating hardware-software integrated mitigations. These techniques can mitigate latest attacks and are more flexible than pure hardware techniques. However, they still have a performance overhead (although better than pure software approaches) and require changes to hardware and OSs.

Gruss et al. proposed KAISER [19], a kernel isolation technique to close hardware side-channels on kernel address information on Intel Skylake processors, such as double page fault attacks or prefetch side-channel attacks. It has low memory and runtime overheads.

Liu et al. proposed CATalyst [27] to defeat cache-based side-channel attacks targeting the LLC in a cloud environment. They leverage a recent feature in Intel processors, called Intel Cache Allocation Technology (CAT), which is part of Intel Cache QoS technologies (that are detailed in chapter 3). Indeed, they use CAT to create two LLC partitions: a non-secure partition (which remains handled by hardware) and a secure partition (which is much smaller and is handled by the hypervisor). Secure pages are assigned to VMs upon request at VM launch time. They offer two security guarantees:

- Cache lines corresponding to secure pages will not be evicted by malicious code.
- There will not be any overlapping of secure pages between different active VMs.

Although the implementation appears to be relatively simple, it requires many precautions to be truly secure. This approach has good performances (especially compared to page coloring techniques) due to the use of hardware features. Furthermore, it is oblivious to the Intel LLC complex addressing scheme. It is also compatible with huge pages and memory deduplication. However, this approach requires changes to the hypervisor, the guest OS and security-critical applications (e.g. new hypercalls and system calls).

2.3.1 Side-channel Attacks Detection Using Hardware Performance Counters (HPCs)

Several detection approaches take advantage of a processor feature called **Hardware Performance Counters** (HPCs). HPCs are special platform-dependent hardware registers [21] [49] used to store statistics about various CPU-related events (clock cycles, retired instructions, cache hits and misses, page faults, context switches, etc.). They are managed by the Processor Monitoring Units (PMUs) in Intel systems.

For instance, HPCs can be used to profile programs in order to understand how they could be optimized. Several efforts have been made to take advantage of HPCs to detect attacks like Rowhammer[10] [37], Return-Oriented Programming [21] and cache-based side-channel attacks, as shown below. HPCs can also be set to interrupt the processor if a given condition is met.

When the processor supports HPCs, the Linux kernel provides an interactive interface to them with the `perf_events` subsystem and the corresponding `perf` tool for userspace (especially `perf-stat`).

Chiappetta et al. [8] proposed a detector based on HPCs. However, due to the limited resolution of `perf-stat`, they developed their own tool `quickhpc` using the Performance Application Programming Interface (`PAPI`) library to probe HPCs. Then

[10]The Rowhammer attack consists in accessing specific memory locations in DRAM with a high repetition rate in order to cause random bit flips in adjacent memory rows, possibly leading to security exploits.

they leveraged machine learning techniques to operate in a more fine-grained manner and therefore detect a spy with more confidence. They simultaneously monitored both the victim and the spy, checking how similar the number of LLC accesses[11] over time was. This is a correlation-based approach. The detector can also use anomaly detection (with a Gaussian distribution, fitting a model for each kind of spy process implementation and then considering all other benign processes as anomalies) as well as supervised learning with neural networks. The resulting detector is a userspace process and has good performances. However, a smarter spy process can inject noise to evade the detector based on correlation.

Zhang et al. [51] present CloudRadar for real-time side-channel attacks detection in clouds. It uses signatures and Dynamic Time Warping (DTW) to detect when a VM is running a cryptographic application (through the Fisher Score to test the repeatability and uniqueness of events selected to identify such applications). Then, the detector uses anomaly detection: when a victim is executing cryptographic applications, it is expected that the cache misses rate be higher than normal for the attacker (in the case of a *Prime+Probe* attack) or that cache misses occur more frequently for the attacker (in the case of a *Flush+Reload* attack). Consequently, the detector does not require changes to the hardware, the hypervisor or guest VMs. It is also extensible to other cloud models such as PaaS (or to non-virtualized environments) and to other side-channel attacks. However, it is very sensitive to noise. Moreover, its three modules require an exclusive use of one physical CPU core and the detector is impacted by the low number of HPCs in current processors.

Payer presents HexPADS [34] to detect cache-based side-channel attacks. It is an HIDS[12] that uses per-attack-vector signatures (generalizing signatures to all attacks in an attack class whenever possible) and leverages existing process metrics and HPCs to collect information about all running processes. It is divided into a collection mechanism, which is the core of the HIDS, and the detector itself which can be switched thanks to a plugin interface. It detects Rowhammer attacks, some cache-based side-channel attacks and CAIN (Cross-VM ASL INtrospection [4]). However, it has a huge Trusted Computing Base (TCB) and uses a very trivial detection based on arbitrary thresholds.

2.3.2 Moving Target Defense

Moving Target Defense (MTD) is a relatively recent mitigation approach against various classes of attacks. It is particularly interesting in the case of side-channel attacks because it addresses the root cause of such attacks: co-location on the same physical machine.

Zhuang et al. [53] tried to formulate a theory of MTD. They explain that although the static nature of information systems is an advantage for maintenance, it also gives attackers a huge advantage: *time*. They point out that this can be tackled with two types of adaptations: movement and transformation. The main idea of MTD is to enlarge the exploration surface and move the attack surface to force the attacker to re-explore the exploration surface. An MTD approach can be used as a reaction, for instance if it incorporates feedback from IDSs and takes advantage of adaptive hardening approaches (cf. Figure 2.9). Zhuang et al. propose lots of definitions (configurable system, operational and security goals, adaptation, configuration space, MTD Problem, configuration entropy, Timing Problem, etc.) in order to start the

[11]Even if data is not present in the CPU cache, each access to it will be registered as an access (or reference) to the LLC

[12]Host Intrusion Detection System

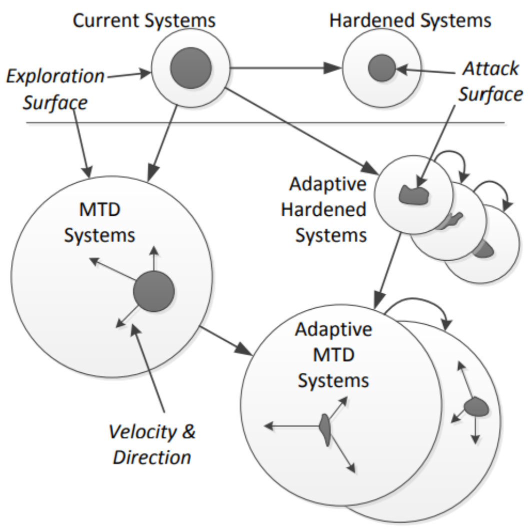


FIGURE 2.9: MTD High Level Intuition [53]

process of developing a MTD Systems Theory, based on configuration management theory. Such a theory will need to be completed with an Attacker Theory to obtain a true MTD Theory.

Moon et al. [31] provide a real implementation of the MTD approach on OpenStack, named Nomad, in order to focus on the root cause of side-channel attacks. This allows Nomad to be agnostic to the specific side-channel vector used and robust against unforeseen side-channels that meet certain conditions. It is scalable and very general in terms of threat model. Indeed, Nomad considers *replication* (e.g. sensitive data can be distributed among several victim VMs) and *collaboration* (e.g. two VMs belonging to the same attacker can cooperate and synchronize in order to be more efficient). Furthermore, it makes no assumption on which clients or VMs are likely to represent threats. Moon et al. propose a VM placement algorithm to solve the optimization problem using heuristics in order to ensure scalability. Their Migration Engine is implemented in the Nova-Scheduler at the Controller node. Nomad is vector-agnostic and addresses a strong adversary model. However, it is not resilient against fast side-channel attacks.

Chapter 3

Proposed Approach

In this chapter, we present the reasoning behind the development of our prototype. First, we discuss the limits of existing mitigations. Then we describe a new Intel technology, named Cache Monitoring Technology, that we want to leverage for detection purposes. Finally, we describe our prototype, including its architecture, its implementation and its deployment.

3.1 Limitations of Existing Detection Solutions

First of all, there is a relatively small number of papers about detection of cache-based side-channel attacks. There are even less of them if we set aside papers about detection of cache-based covert channels and particularly if we only focus on detection in clouds.

Besides, many detection solutions and countermeasures presented in Chapter 2 only protect against a subset of cache-based side-channel attacks. Furthermore, most of them cause significant overheads or are not scalable, and require important modifications to existing deployments.

The detector proposed by Chiappetta et al. [8] is based on HPCs but we believe that it does not take full advantage of them. Moreover, it is only a detector because no mitigation is implemented in case of an attack being detected. As pointed out by Gruss et al. [17], the detector would have a high false positives rate as soon as an application runs intensive calculations. Gruss's detector partly corrects this issue but we believe that a new Intel technology named Cache Monitoring Technology (CMT) can improve the detection. Furthermore, both detectors mainly focus on attacks carried out between processes. We aim at specifically detecting attacks in a virtualized environment.

3.2 Intel's Cache Monitoring Technology

As already pointed out, processor caches are central to performance optimization. Consequently, there have been efforts to bring cache Quality of Service (QoS) to modern processor architectures. However, existing techniques were mostly heuristics, especially because there was no methodology to identify cache lines belonging to a particular thread. Furthermore, these techniques lacked configurability by the operating system.

In order to address these issues, a QoS framework [35][38] was recently introduced into the Linux kernel by Intel developers. It is mainly an abstraction layer to a hardware feature set provided by Intel and called Intel **Resource Director Technology** (RDT) [22].

3.2.1 Intel Resource Director Technology

There are two types of RDT features: **monitoring** capabilities and **allocation** (resource control) capabilities.

There are two different RDT monitoring features:

- **Cache Monitoring Technology** (CMT): It allows a system management agent (e.g. an operating system or an hypervisor) to determine the usage of cache by applications running on the platform. The initial implementation focuses on L3 cache monitoring (which currently corresponds to the LLC in most machines).
- **Memory Bandwidth Management** (MBM): It was introduced later and uses the same infrastructure as CMT to permit the monitoring of bandwidth from one level of the cache hierarchy (including the main memory) to the next. As with CMT, the initial implementation focuses on the L3 cache and thus MBM monitors system memory bandwidth. There are two modes for MBM: the *total external* mode which monitors the L3 total external bandwidth to the next level of the cache hierarchy (including prefetch misses) ; the *local* mode which monitors the L3 external bandwidth satisfied by the local memory. The difference between these two modes only has meaning for a non-uniform memory architecture (NUMA). In this case, it allows tracking the bandwidth to off-package memory resources (e.g. to memory on another physical processor).

As for RDT allocation features, there are three different features:

- **Cache Allocation Technology** (CAT): It allows a system management agent to specify the amount of cache space that an application can use, and to a certain extent the location of this space (typically by using bitmasks to specify cache ways). The initial implementation allowed those actions on L3 caches but L2 caches are now also supported.
- **Code and Data Prioritization** (CDP): It allows a system management agent to specify different amount of caches that an application can use to load its code and its data. It is essentially an extension of CAT. It distinguishes code (i.e. processor instructions) from data, enabling separate prioritization of code and data fetches to L3 caches.
- **Memory Bandwidth Allocation** (MBA): MBA provides control over memory bandwidth available per-core or per-thread. It is roughly to MBM what CAT is to CMT. It also uses the same infrastructure as CAT.

Because of our main objective of detecting cache-based side-channel attacks, we focus on RDT monitoring features and more particularly on CMT.

3.2.2 RDT Monitoring Features, CMT and RMID

Monitoring Infrastructure

The key component of the RDT monitoring infrastructure is the **Resource Monitoring ID** (RMID). The RMID is a software-defined ID which references applications or VMs that are scheduled to run on a core. This ID is used by CMT and MBM. In both cases, the infrastructure provides the following mechanisms:

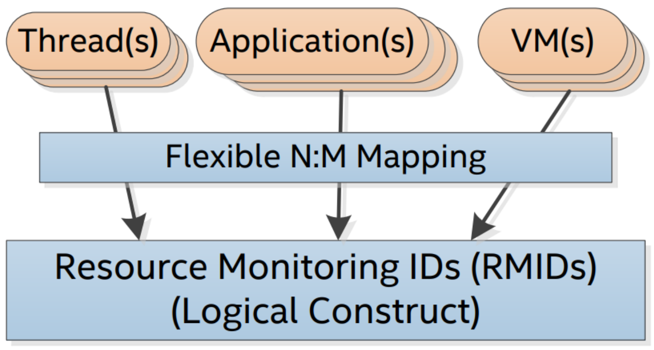


FIGURE 3.1: High-level view of RMID mapping [35]

- Detect which monitoring capabilities are supported by the platform. This is achieved through a combination of `CPUID`[1] instructions.
- Allow the OS or the hypervisor to attribute RMIDs to applications or VMs.
- Monitor cache occupancy or memory bandwidth on a per-RMID basis.
- Allow the OS or the hypervisor to read LLC occupancy or memory bandwidth for a given RMID at any time.

From now on, we only consider how CMT works with an hypervisor and VMs (which are basically groups of threads).

RMID Management by Hypervisors

Once the hypervisor has verified that the platform it runs on supports CMT, it can associate an RMID with one or several VMs. This is accomplished through the writing of a *Model Specific Register* (MSR) named `IA32_PQR_ASSOC`. As illustrated in Figure 3.2, this core-specific MSR contains the RMID but also a CLass Of Service (CLOS). The CLOS is like the RMID except that it is used by RDT allocation features (e.g. CAT). Obviously, the number of RMIDs available on a system is limited by the size of the corresponding field in the `IA32_PQR_ASSOC` MSR, and that is part of the parameters that the software layer can discover with `CPUID` instructions.

The process of associating a RMID to VMs needs to be done at every context switch by the hypervisor. Writing to an MSR can be done with the `WRMSR` processor instruction. Once the association is established, the software may execute for a period of time while the hardware tracks cache occupancy for the RMID currently contained in the core's `IA32_PQR_ASSOC` MSR. Indeed, every memory request is *tagged* with this RMID, allowing the hardware to do the accounting.

At anytime, the hypervisor is responsible for keeping an N:M mapping between currently assigned RMIDs and VMs (cf. Figure 3.1). Then, whenever it needs, the software can decide to retrieve the occupancy value for a given RMID (thus corresponding to one or several VMs) through an interface composed of two other MSRs. The first MSR is named `IA32_QM_EVTSEL`[2] and the hypervisor is expected to write to it in order to specify the RMID and the event ID it is interested in. In the current implementation, the event ID can correspond to CMT, MBM local or MBM

[1] An instruction for the x86 architecture allowing software to enumerate and detect functionalities supported by the processor

[2] Also named `IA32_QOSEVTSEL`

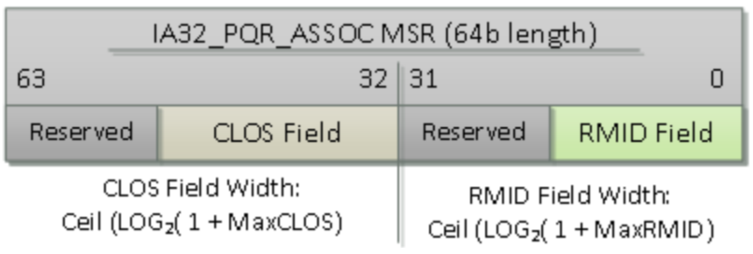


FIGURE 3.2: `IA32_PQR_ASSOC` MSR [35]

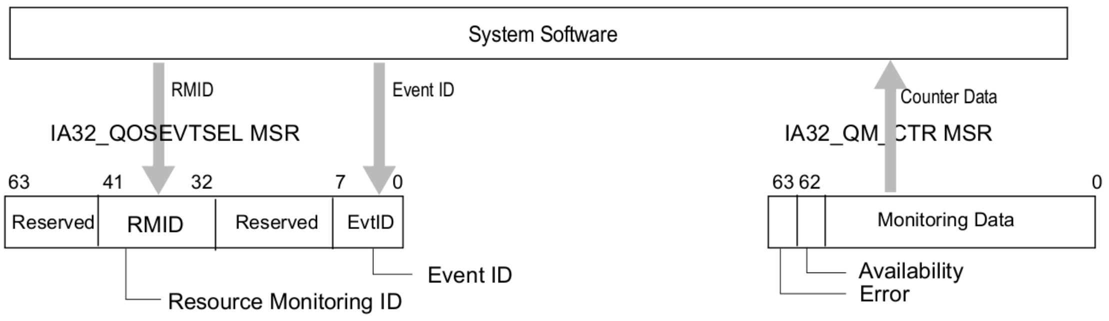


FIGURE 3.3: Usage of `IA32_QOSEVTSEL` and `IA32_QM_CRT` MSRs [22]

total. The hardware returns the value using the second MSR, `IA_32_QM_CTR`, which the hypervisor can read using the `RDMSR` processor instruction. Two bits of this MSR are reserved for signaling availability of data and potential errors: if both are not set, the returned data can be considered valid by the software. All of this is summarized in Figure 3.3.

In the case of cache occupancy (i.e. when using CMT), the data returned in the MSR may be optionally multiplied by an upscaling factor in order to convert it to bytes. This factor can be fetched using `CPUID`. This will be discussed further in this report when presenting our prototype.

3.2.3 Interfaces to Intel RDT Features

Several efforts have been made to develop interfaces to Intel RDT features.

Intel developed a software layer, `intel-cmt-cat`[3], which supports all Intel RDT features. However, allocation and monitoring features can only be used at core-granularity. Indeed, the software lacks support from the operating system to work at task-granularity. For instance, in the case of CMT, this means that a RMID is statically assigned to a CPU core. This approach is only suitable for a static configuration (e.g. applications are pinned to CPU cores) since a thread migrating across CPU cores will not be tracked. This approach is known as **standalone cache monitoring** [32].

Another approach is **scheduler-based cache monitoring** that involves the operating system scheduler. This allows software to manage RMIDs and then to expect the scheduler to associate the core with the correct RMID when the corresponding thread is to be scheduled on that core. The scheduler is also in charge of managing RMIDs when a thread is de-scheduled or migrated to a different core (or even to a

[3]`https://github.com/01org/intel-cmt-cat`

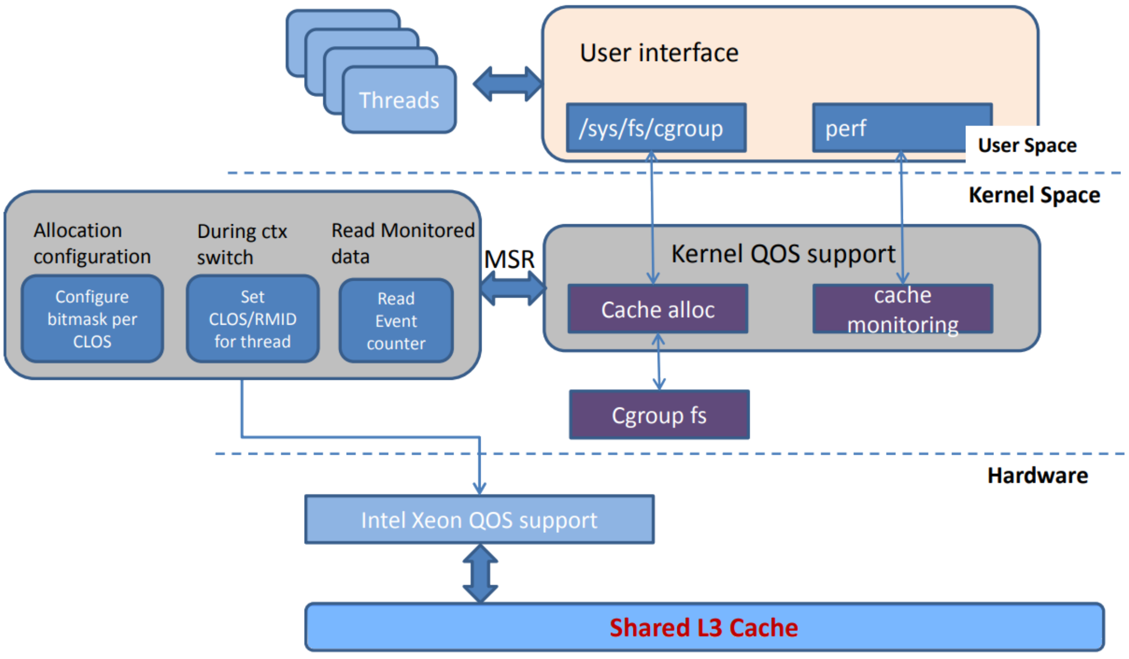


FIGURE 3.4: Linux kernel implementation of Intel cache QoS [38]

different processor socket, which involves RMID remapping since RMIDs are local to a socket).

Consequently, in the Linux kernel, support for Intel RDT monitoring features is based on the `perf_event` kernel subsystem[4]. This subsystem provides a framework for performance analysis in the Linux kernel, including support for HPCs and CMT.

To this end, the `perf_event` subsystem can setup callbacks in the scheduler, allowing it for instance to execute actions at every context switch. It can then select the RMID associated with the thread being scheduled and assign it to the CPU core by writing to the aforementioned MSR (cf. Figure 3.4). This causes following memory requests and related cache fetches to be associated with this RMID. A special default RMID is associated with threads which are not to be monitored (e.g. `0` in Xen implementation).

When a thread terminates and its RMID is not associated with any other thread, it is "released" and put back into a pool of unused/available RMIDs. The `perf_event` subsystem can also handle inheritance and thus assign to child processes the RMID of their parent. The `perf_event` subsystem can be controlled from userspace using the `perf_event_open(2)` system call (which we used in our prototype and is thus detailed in next section) or the higher-level `perf` tool.

Intel RDT Support in Common Hypervisors

Intel RDT features and especially monitoring features are also supported by hypervisors.

[4]However, even if it does not disable it, `perf` corrupts CAT and CDP configuration. Therefore, support for RDT allocation features started with a *cgroup*[5] interface for L3 CAT. However, it remained an out-of-tree patch. The true implementation was introduced in kernel 4.10 as `resctrl` (*resource control*) and added support for L2 CAT and L3 CDP (MBA is not fully supported yet). The `CONFIG_INTEL_RDT_A` Kconfig option needs to be enabled.

[5]Basically, cgroups are similar to traditional groups of processes except that they are organized into a hierarchy.

- KVM directly benefits from the `perf_event`-based implementation of RMID management in the Linux kernel. Indeed, KVM makes the Linux kernel an hypervisor and thus the Linux scheduler can manage VMs as it manages traditional processes.
- On the contrary, Xen requires scheduler enhancements to support Intel RDT monitoring features. It started with Xen 4.5 supporting CMT and as of Xen 4.7, CAT, MBM and CDP are also supported. They can be managed using the XL software layer [42]. Dario Faggioli pointed out some issues [11] with the current implementation of CMT, such as the limited number of RMIDs, RMID association asynchronousness and per-vCPU cache monitoring.

3.2.4 CMT Use Cases

The primary purpose and usage of CMT is to trace history of cache usage for processes or VMs. This is useful to identify cache-sensitive applications, i.e. applications whose performances are related to cache occupancy. Finding the *optimal cache operating point* is then an interesting way of profiling such applications.

Xen started using and leveraging CMT in its scheduler in order to achieve high-level load-balancing [11].

In our prototype, we leverage CMT as a mean of detecting cache-based side-channel attacks targeting the LLC, which is shared between VMs running on the same CPU socket.

3.3 Prototype Description

In this section, we describe the objectives of our prototype and detail its architecture, its implementation and how it can be deployed.

3.3.1 Objectives and Threat Model

Objectives

We aim at developing a prototype that detects cache-based side-channel attacks in a virtualized environment. This means that we focus on attacks between VMs. VMs in public clouds are usually running on different cores, *i.e.* there cannot be two VMs running on the same core. Therefore, cache-based side-channel attacks target the LLC cache which is shared between CPU cores and thus between VMs.

The primary goal is to study whether Intel CMT can be used to improve existing detection methods. Due to previous work on such detection methods, we choose to leverage HPCs and anomaly detection. There are a number of techniques for anomaly detection. We choose to experiment with Gaussian distributions. This will be detailed in the implementation section.

Since CMT allows monitoring cache occupancy, we believe that the *Prime+Probe* technique is a good candidate for being detected. Indeed, it is expected to cause relatively important cache occupancy variations because of eviction sets (each representing several cache lines). On the contrary, *Flush+Reload* only flushes a few cache lines. *Flush+Flush* is even more challenging because it causes much less cache misses and is thus less likely to be detected with HPCs.

Threat Model

We consider a threat model with VMs co-located on the same physical machine. **One** malicious VM attacks **one** target VM. We do not deal with two or more attackers targeting the same victim (deliberately or not).

Ideally, we want to detect attackers and not victims (even if they often have a similar behavior during a cache-based side-channel attack) because the latter should not be disturbed by our module. Another advantage of targeting attacker VMs is that there are generally more victims than attackers, thus less suspect VMs are to be moved. Furthermore, we can afford some false positives since we are only going to "live-migrate" suspect VMs (following the MTD approach).

Our detector runs directly on the physical host machine, i.e. as a process running on the hypervisor. Thus, we consider that it cannot be disturbed from inside a VM, except through a potential denial-of-service *via* cache-based side-channel attacks.

3.3.2 Architecture

Our detector is a standard application running as a Linux process. Indeed, we choose to work with the QEMU-KVM hypervisor[6]. Because it uses the Linux kernel scheduler, KVM handles CMT resource association and resource marking for us. Then, from the host, VMs are managed like groups of QEMU threads and can be monitored using Process IDs (PIDs).

The detector takes advantage of the `libvirt` library: it is an open source API that can be leveraged with C, but also a daemon and management tool for interfacing with various hypervisors such as QEMU-KVM and Xen. We choose to use `libvirt` because the provided abstraction layer has several benefits, such as:

- It makes the MTD response phase easier thanks to the migration API.
- It makes our detector easier to use with other hypervisors than QEMU-KVM.
- It will help extend the detection to several physical machines connected through the network.
- It facilitates error handling.

However, `libvirt` voluntarily hides some details in order to provide stronger abstraction. One of them is the PID of running guests and as detailed in the implementation section, we need to fetch this information bypassing `libvirt` and thus this will need to be adapted if our detector is intended to run with other hypervisors than QEMU-KVM. Likewise, the API provided by `libvirt` for `perf_event` is still under development and lacks many events that we are interested in. Therefore, we also had to bypass this part.

Our detector process contains three threads:

- The **main thread**: it is the process start-up thread and it is responsible for periodically probing (i.e. fetching HPCs and CMT values) all the running VMs.
- The **guests update thread**: it periodically scans the host to maintain a linked list of running VMs.

[6]The relation between QEMU and KVM to form an hypervisor is reminded on Figure A.1 in appendix A.

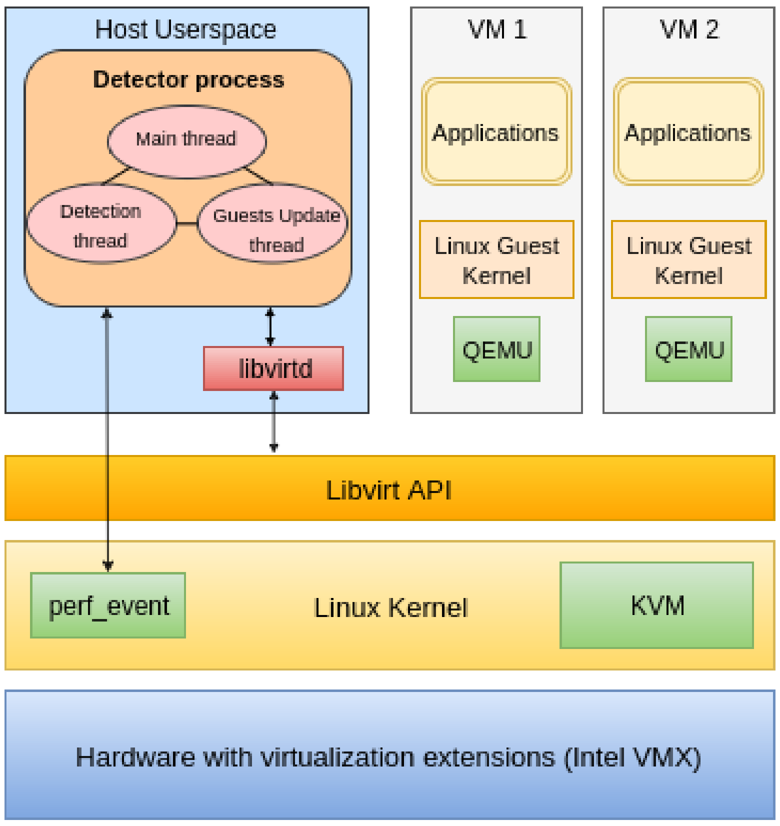


FIGURE 3.5: Architecture of our detector

- The **detection thread**: it periodically runs the anomaly detection algorithms and launches the MTD response when a possible cache-based side-channel attack is detected.

The architecture of our detector is summarized in Figure 3.5.

3.3.3 Implementation of our Prototype

The detector is entirely developed in C. Indeed, we needed to have a low-level access to the `perf_event` subsystem and directly using the `perf_event_open` system call seemed easier and cleaner. The main drawback was that the anomaly detection part would have been easier to develop using Python's machine learning libraries.

Thread to Update List of Guests

The guests update thread runs periodically to maintain a linked list of running VMs. This is achieved using the `libvirt` API. However, we need the PID of the QEMU instance corresponding to each VM in order to collect `perf` events. Unfortunately, `libvirt` voluntarily makes this information unavailable to prevent API users from inadvertently interfering with the library's internals. Therefore, we had no choice but to implement a function walking the `procfs` pseudo-filesystem in order to fetch the PID of each QEMU instance corresponding to a running `libvirt` guest. Then, a guest is represented through encapsulated structures containing all the information needed by the detector:

`include/kvm.h`

```
[...]

struct perf_event {
        perf_event_id id;
        int fd;
        unsigned int attr_type;
        unsigned long long attr_config;
        union {
                /* cmt */
                struct {
                        int scale;
                } cmt;
        } efields;
};

struct perf_samples {
        struct perf_event *perf_event;
        int ringloc; /* to replace LRU value */
        double records[NR_SAMPLES];
};

struct kvm_guest {
        int pid;
        virDomainPtr libvirt_dom;
        enum perf_status {
                IGNORE,
                COUNTING,
        } perf_status;

        /* File descriptor of perf events group leader */
        int group_fd;
        /* Perf events collected samples */
        struct perf_samples *perf_samples[NR_EVENTS];

        /* pointers to next and previous kvm guests in list */
        struct kvm_guest *prev;
        struct kvm_guest *next;
};

[...]
```

In particular, each guest keeps as many sliding windows (basically arrays of values) as there are monitored events. The size of this sliding window (i.e. how many last values are kept) can be configured *via* a command-line option when launching the detector.

The main loop composing the guests update thread is implemented in the `kvm_scan_libvirt()` function. Algorithm 1 shows the pseudocode of this function.

Main Thread

The main thread is responsible for **periodically** probing the monitored events. These events are directly configured in the source code but can be easily changed thanks to helping macros. The `perf_event` subsystem is used through the `perf_event_open` system call. When a VM starts running and before it is added to the linked list, this system call is called for each monitored events (whether it is HPCs or CMT). It returns a file descriptor which is kept and can be read from when probing for new values.

Algorithm 1: Pseudo-code for `kvm_scan_libvirt()` function

```
Input: Current sorted list $l$ of monitored VMs
Output: Updated sorted list $l$ of monitored VMs
$activeIDs \leftarrow$ list of active guests IDs returned by libvirt;
for $i \in activeIDs$ do
    if an element in $l$ has ID $i$ then
        remember guest as still present;
        go to next iteration;
    else
        create structure representing new KVM guest of ID $i$;
        initialize perf counters for new guest;
        add new guest to $l$;
    end
end
foreach element $m \in l$ that was not accessed do
    remove $m$ from $l$;
    delete $m$;
end
```

The probe delay (*i.e.* the time period between two consecutive probing of all `perf` events) can be configured *via* a command-line option when launching the detector. This resolution is limited: 10 ms was measured to be working but 1 ms was too low[7].

Due to hardware limitation, only 4 HPCs can be enabled simultaneously. Otherwise, multiplexing is required but this decreases the resolution and consequently the accuracy. CMT is accessed using the `perf_event_open` system call but it is not an HPC. This is why we can use up to 5 *raw features* for anomaly detection (one of them being CMT).

The `perf_event_open` system call expects a structure containing many configuration options, including the PID of the monitored process. In our case, this PID is the PID of the QEMU process corresponding to the VM. In particular, we enable inheritance, which means that all children of the monitored process will also be monitored. This ensures that all threads corresponding to a given VM will be accounted for. Indeed, a QEMU instance is composed of one thread per vCPU in addition to one thread for IOs.

`libvirt` also implements an abstraction layer for `perf` events but only a few are accessible for now and it lacks flexibility. However, looking at the `libvirt`'s source code responsible for implementing `perf` events helped us choose certain options for the `perf_event_open` system call.

The main loop composing the main thread is implemented in the `perfctr_probe()` function. Algorithm 2 shows the pseudocode of this function.

Detection Thread

Finally, the detection thread periodically runs the anomaly detection algorithm on the entire list of running VMs. For instance, it can run every second with a sliding window of 100 values if the `perf` events are probed every 10 ms. Its main loop is implemented in the `detector()` function. Algorithm 3 shows the pseudocode of this

[7]This was estimated using the `PERF_COUNT_HW_CPU_INSTRUCTIONS` event corresponding to retired CPU instructions.

Algorithm 2: Pseudo-code for `perfctr_probe()` function

```
Input: Current sorted list $l$ of monitored VMs
foreach monitored VM $m \in l$ do
    foreach monitored event $e$ of $m$ do
        $value \leftarrow$ value of perf event $e$;
        if $e$ is CMT then
            $value \leftarrow value \times CMTscale$;
        end
        replace older value in sliding window of $e$ by $value$;
    end
    reset all counters for $m$;
end
```

function. The anomaly detection technique we implemented is detailed in the next subsection.

Algorithm 3: Pseudo-code for `detector()` function

```
Input: Current sorted list $l$ of monitored VMs
if we are in setup mode then
    store cross-validation set $CV$;
else
    estimate Gaussian parameters $G$ for $l$;
    load cross-validation set $CV$;
    select threshold $\varepsilon$ using $CV$ and $G$;
    foreach monitored VM $m \in l$ do
        compute p($m$, $G$) and compare to $\varepsilon$;
        deduce if $m$ is malicious;
    end
end
```

Anomaly Detection

Over the past few years, machine learning techniques have gotten more attention in information systems security. Anomaly detection is a commonly used type of machine learning application. It can be implemented through various algorithms.

A popular choice is to leverage the Gaussian distribution[8]. For instance, this technique is used by Chiappetta et al. [8] to detect processes performing cache-based side-channel attacks.

In our mitigation approach, we use the Gaussian model to detect malicious VMs. Because a "normal" behavior is difficult to characterize for a VM, we choose to fit the Gaussian distribution to particular cache-based side-channel attacks, thus designing *attack profiles.*

Given a training set $\left\{x^{(1)}, ..., x^{(m)}\right\}$, we want to estimate the Gaussian distribution for each of the x_i features . For each feature $i = 1...n$, we need to find parameters

[8]Also called the normal distribution

μ_i and σ_i^2 that fit the data in the i-th dimension of $\left\{x^{(1)}, ..., x^{(m)}\right\}$ (*i.e.* the i-th dimension of each sample).

The Gaussian distribution is given by

$$p\left(x; \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.1}$$

where μ is the mean and σ^2 controls the variance.

We can estimate the parameters (μ_i, σ_i^2) of the i-th feature using the following equations:

$$\mu_i = \frac{1}{m}\sum_{j=1}^{m} x_i^{(j)} \tag{3.2}$$

$$\sigma_i^2 = \frac{1}{m}\sum_{j=1}^{m}\left(x_i^{(j)} - \mu_i\right)^2 \tag{3.3}$$

Once the estimation of the Gaussian parameters is done, we can investigate which samples have a high probability given this distribution and which samples have a low probability. The high probability samples are likely to be the anomalies in the dataset. One way to determine which samples are anomalies is to select a threshold based on a **cross-validation set**. This means that we use a set[9] $\left\{\left(x_{CV}^{(1)}, y_{CV}^{(1)}\right), ..., \left(x_{CV}^{(m_{CV})}, y_{CV}^{(m_{CV})}\right)\right\}$ where the label $y = 1$ corresponds to anomalous samples (*i.e.* malicious VMs) and $y = 0$ corresponds to normal samples (*i.e.* honest VMs). This is called **supervised learning** because we manually label malicious VMs in the cross-validation set.

For each cross-validation sample, we will compute $p(x_{CV}^{(i)})$. The vector of all these probabilities $p(x_{CV}^{(1)}), ..., p(x_{CV}^{(m_{CV})})$ is passed together with $y_{CV}^{(1)}, ..., y_{CV}^{(m_{CV})}$ to the function responsible for selecting the threshold value. This function returns two values. The first one is the selected threshold ε. If a sample has a probability $p(x) > \varepsilon$ then it is considered to be an anomaly. The function also returns the F_1 score, which represents how well a certain threshold is adapted for accurately differentiating anomalous from normal samples. For many different values of ε, the function computes the resulting F_1 score by computing how many samples the current threshold classifies correctly and incorrectly. The F_1 score is computed using the precision (*prec*) and the recall (*rec*):

$$F_1 = \frac{2 \cdot prec \cdot rec}{prec + rec} \tag{3.4}$$

Precision and recall are computed according to:

$$prec = \frac{tp}{tp + fp} \tag{3.5}$$

$$rec = \frac{tp}{tp + fn} \tag{3.6}$$

where:

- tp is the number of true positives
- fp is the number of false positives

[9]Recorded during an execution of the cache-based side-channel attack we wish to detect

- fn is the number of false negatives

In addition to the traditional Gaussian model, the *multivariate* Gaussian model is also used for anomaly detection purposes. It is a slightly different technique that is supposed to catch some anomalies which cannot be detected with the non-multivariate Gaussian distribution. In this method, $p(x)$ is calculated according to the following equation:

$$p\left(x;\mu,\Sigma\right) = \frac{1}{(2\pi)^{\frac{n}{2}}\,|\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{3.7}$$

where:

- μ is the mean (n-dimensional vector)
- Σ is the covariance matrix

These parameters can be estimated using the following equations:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \tag{3.8}$$

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu)(x^{(i)}-\mu)^T \tag{3.9}$$

We choose to experiment with the two models (or *modes*).

To sum up, we implement the following two modes for Gaussian-based anomaly detection:

- The *original* model: it requires manually creating combinations of "raw" features (i.e. parameters) to capture complex anomalies. It is computationally cheaper and thus scales better.
- The *multivariate* model: it can "automatically" detect and take advantage of correlation between features. However, the size of the training set needs to be much larger than the number of features.

To implement the anomaly detection module of our prototype, we use the GNU Scientific Library[10]. Two modes (*i.e.* original and multivariate) can be selected through a command-line option when launching the detector. A first run during a cache-based side-channel attack is necessary in order to create the cross-validation set (i.e. `perf` values together with labels identifying malicious VMs). We create this cross-validation set and then manually assign a label to each sample. This set will be used to select a threshold (ε) using the F_1 score in order to determine which samples from the training sets are anomalies.

CMT Granularity

During the implementation of our prototype, we noticed that CMT had a high granularity. Indeed, the returned value was a multiple of **32768** bytes. This value of 32768 is read from the `/sys/devices/intel_cqm/events/llc_occupancy.scale` file by our program and corresponds to a value initialized by the Linux kernel implementation of CMT. After reading the related Linux source code in `arch/x86/events/intel/cqm.c`

[10] https://www.gnu.org/software/gsl/

and `arch/x86/kernel/cpu/common.c`, we confirmed that CMT was indeed implemented with a 32 kB granularity. This is obviously an important limitation for our use case.

3.3.4 Deployment of the Detection Agent

We developed our prototype on a test server with an `Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz` processor supporting Intel CMT. It runs Debian Jessie (8.8) but not with the default Linux kernel. Indeed, CMT and MBM require at least the 4.6 kernel. Therefore we used the latest LTS kernel, i.e. the 4.9 stable branch.

We enable KSM[11] for the host and guests machines, and we use the `libvirt-dev` package to have a full access to functions provided by the `libvirt` API.

Nevertheless, reproducing a cache-based side-channel attack is very difficult. We managed to carry out simple (i.e. that do not recover private keys) *Flush+Reload* and *Flush+Flush* attacks. However, we expect to detect *Prime+Probe* attacks and this technique is more complex, especially the construction of the eviction sets.

We were close to achieve that with Mastik [50][28] but although it was working well for the attacks between two threads, there were several issues to reproduce such attacks between two VMs. Thus, we contacted its author, Yarom Yuval, and narrowed the problem down to QEMU-KVM causing a VM-Exit on `RDTSC` instructions (maybe for TSC frequency scaling). We were not able to find a stable way of modifying this behavior. Therefore, to overcome this issue, we used IAIK's *Prime+Probe* implementation [16] after having applied a few patches to it.

[11]Kernel Same-page Merging, *i.e.* Linux implementation of page deduplication

Chapter 4

Experimental Results

In this chapter, we present our results and interpret them in order to evaluate our prototype and our approach to detect cache-based side-channel attacks taking place between two VMs.

In order to employ a machine learning method, a large number of training sets is required to obtain meaningful results. As we use VMs with only one vCPU, each guest instance represents two threads on the host system (one thread for the vCPU and one thread for IOs). To run the detector agent, we also need two cores allocated to the agent. Therefore, as we have one processor socket with 8 physical cores and hyperthreading enabled, we choose to run 6 VMs: each VM (with its two corresponding threads) is pinned to a physical CPU core. For that purpose, we leverage processes CPU affinity using the `taskset` command-line utility.

Our experiments were performed on the following platform:

Table 4.1: Specifications of the test platform

Product	SuperServer 5028D-TN4T
Processor	`Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz`
Number of sockets	1
Number of cores per socket	8
Number of threads	16
System memory	ECC DDR4 SDRAM 1 x 16 GB
L1d cache	32 kB
L1i cache	32 kB
L2 cache	256 kB
L3 cache (LLC)	12288 kB

4.1 CMT Behavior During the Attack

Figure 4.1 shows how CMT varies for an attacker and a victim VM when a *Prime+Probe* attack [16] is launched (at `t = 4 s`). During the experiment, the victim VM is running a cache-sensitive application.

We see that the LLC occupancy of the attacker VM clearly increases when the attack starts and is then stable. This corresponds to the creation of the eviction sets, which are periodically probed by the attacker program.

Moreover, the LLC occupancy of the victim VM seems constant. This is expected due to the granularity of CMT, as only some cache lines of the victim VM are periodically evicted by the attacker before being reloaded and thus are not measurable.

However, CMT variations are less clear when comparing between several VMs. Indeed, some idle VMs sometimes show a higher LLC occupancy than attacker or victim VMs, as it will be apparent in the next tables.

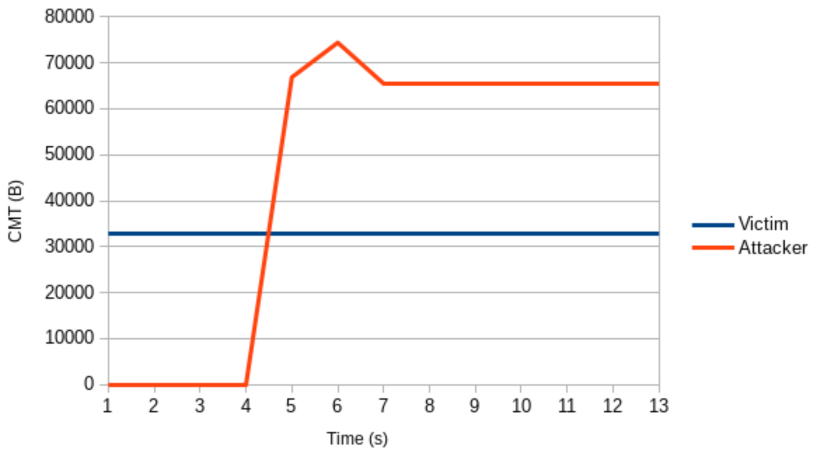


FIGURE 4.1: Variation of CMT value during a *Prime+Probe* attack

4.2 Some Experimentations

We want to evaluate our detector in order to conclude on the efficiency and added value of CMT-based detection of cache-based side-channel attacks. We thus base our work on prior use of HPCs for detection of cache-based side-channel attacks proposed by Chiappetta et al. [8] and Gruss et al. [17]. We want to compare the detection accuracy of our detector, *i.e.* its ability to correctly distinguish honest and malicious VMs.

Gruss et al. [17] experimentally chose the following HPCs for their detector:

- **LLC misses**: This counter indicates cache misses in the LLC. It is relevant for cache-based side-channel attacks detection because such attacks (including *Prime+Probe*) cause a lot of cache misses due to cache lines eviction (cf. section 2.2).
- **LLC references**: This counter indicates LLC accesses (including prefetches). It is interesting when used together with the above counter to represent cache miss rates.
- **ITLB read and write accesses**: These two counters represent the number of accesses to the instruction TLB cache. This is useful to *normalize* the two previous counters in order to differentiate cache-based side-channel attacks from applications running intensive calculations. Indeed, the main loop of cache-based side-channel attacks causes a high number of cache accesses and cache misses while executing only a small piece of code, thus executing only a low pressure on the ITLB.

For some reason, the `perf` events corresponding to ITLB write accesses and write misses do not work on our test server. Therefore, we only use the other HPCs. Tables 4.2, 4.3 and 4.4 show the mean values for those events. The sliding window is of size 100[1] and events are probed each 10 ms. We want to check whether adding CMT as a feature for anomaly detection improves the efficiency of the proposed approach.

[1]It means that the last 100 values are kept in memory for each `perf` event of each running VM.

Table 4.2: Mean values for idle VMs

VM	**CMT**	**LLC references**	**LLC misses**	**ITLB read accesses**
1	32768	18.96	18.96	0.06
2	0	19.38	19.38	0.09
3	65536	19.22	19.22	0.05
4	98304	16.13	16.13	0.04
5	65536	31.85	31.85	0.27
6	61932	20.08	20.08	0.03

The above values are rather expected. It should be noticed that even if all VMs are in the idle state (except VM 1, which is running the victim program), some of them have a non-zero cache occupancy (because of *daemons* and kernel processes).

Table 4.3: Mean values for *Prime+Probe* attack on VM 1 from VM 2

VM	**CMT**	**LLC references**	**LLC misses**	**ITLB read accesses**
1	32768	19.81	19.81	0.04
2	125501	17.39	17.39	0.20
3	108789	18.02	18.02	0.02
4	36045	19.83	19.83	0.13
5	65208	17.71	17.71	0.11
6	65536	13.71	13.71	0.02

During the *Prime+probe* attack [16] between VM 2 and VM 1, we observe that the LLC occupancy for the attacker VM has significantly increased. Although the LLC occupancy has also increased for VM 3 (which still is in the idle state), it remains lower than for the attacker VM. Figure 4.2 highlights this behavior.

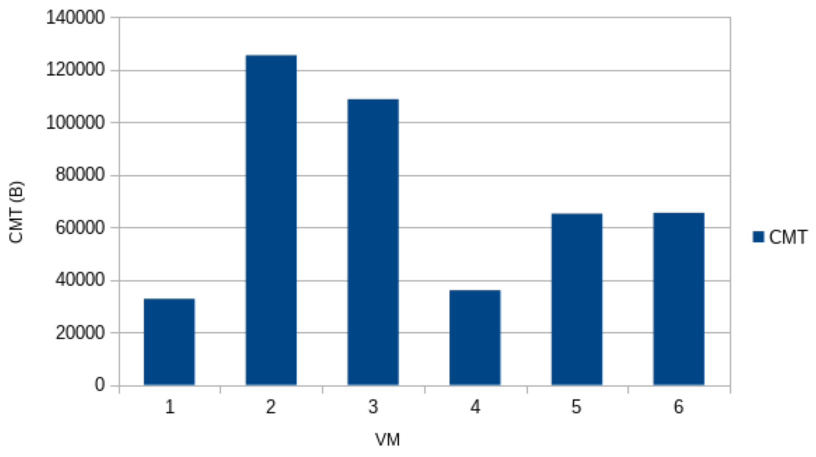


Figure 4.2: CMT values for all running VMs during a *Prime+Probe* attack

We also remark that the number of LLC misses has increased for the victim VM. This was expected because some of its cache lines are periodically evicted by the attacker VM. However, we notice that the mean value for LLC references equals the mean value for LLC misses. This can be explained by our use of mean values on sliding windows. It should also be noticed that the number of ITLB read accesses has increased for the attacker VM because of the aggressive attacker program. Figure 4.3 highlights these behaviors.

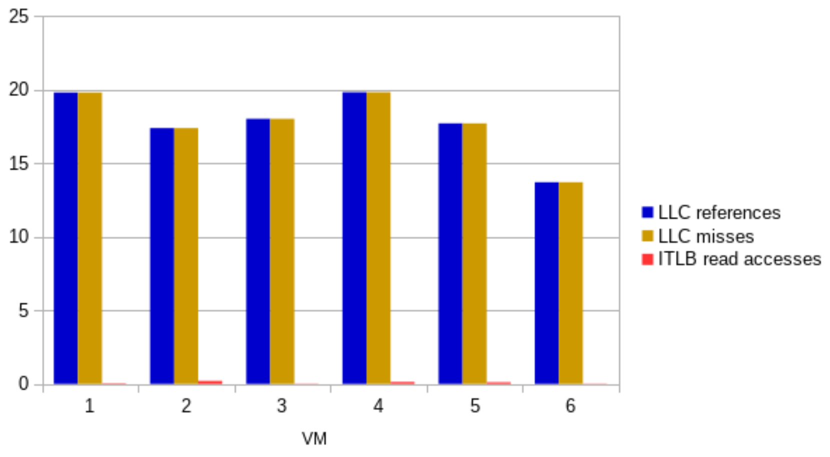


FIGURE 4.3: Other `perf` events values for all running VMs during a *Prime+Probe* attack

TABLE 4.4: Mean values during an execution of the `stress` command on VM 2

VM	CMT	LLC references	LLC misses	ITLB read accesses
1	0	19.47	19.47	0.30
2	212337	13393	13395	417.48
3	0	18.06	18.06	0.32
4	0	19.17	19.17	0.05
5	0	18.00	18.00	0.06
6	0	17.86	17.86	0.04

In the last case, we run the `stress` command. It is a tool that stress tests a system (acting on CPU, memory, I/O and disks). We use it to generate a rather high activity on one VM to simulate a random load and test our detector's ability to not fall for VMs running intensive computations. We observe as expected that all considered events return high values.

Combinations of Raw Features

Gruss et al. [17] suggest using combinations of these features. They propose to use ITLB-related events to normalize other events in order to differentiate cache-based side-channel attacks from intensive computations. Tables 4.5, 4.6 and 4.7 show the mean values for CMT and the resulting two combinations of raw features used by Gruss et al.[2].

[2]Without ITLB write accesses, as explained earlier

Table 4.5: Mean values for idle VMs

VM	CMT	$\frac{LLC_{references}}{ITLB_{\text{read accesses}}}$	$\frac{LLC_{misses}}{ITLB_{\text{read accesses}}}$
1	32768	3.96	3.96
2	0	1.03	1.03
3	32768	4.98	4.98
4	42598	6.86	6.86
5	0	4.06	4.06
6	86835	2.17	2.17

Table 4.6: Mean values for *Prime+Probe* attack on VM 1 from VM 2

VM	CMT	$\frac{LLC_{references}}{ITLB_{\text{read accesses}}}$	$\frac{LLC_{misses}}{ITLB_{\text{read accesses}}}$
1	32768	4.44	4.44
2	123863	0.51	0.51
3	5570	6.08	6.08
4	17695	4.61	4.61
5	0	2.03	2.03
6	13434	2.10	2.10

Our previous remark regarding equality between LLC references and misses is still valid.

We notice that the use of ratios is interesting: the two combinations of features decreases for the attacker VM during the *Prime+Probe* attack thanks to the division by the high value of ITLB read accesses. Conversely, the ratios are still high when executing the stress test:

Table 4.7: Mean values during an execution of the `stress` command on VM 2

VM	CMT	$\frac{LLC_{references}}{ITLB_{\text{read accesses}}}$	$\frac{LLC_{misses}}{ITLB_{\text{read accesses}}}$
1	0	1.84	1.84
2	252641	37.83	37.86
3	0	1.06	1.06
4	0	3.74	3.74
5	1311	9.41	9.41
6	983	3.21	3.21

We also observe that the ratios are lower in this case than during the *Prime+Probe* attack. This was expected because although it trashes the overall LLC, the stress test is not specifically targeting sets used by the victim VM.

To further characterize these observed behaviors, we perform the anomaly detection approach. We store a cross-validation set with VM 5 performing a *Prime+Probe* attack on VM 3. Then we run the detector with the two aforementioned combinations of features, with and without CMT. The attack is this time performed by VM 2 on VM 1. The results are shown in table 4.8.

TABLE 4.8: F_1 score for cross-validation set with and without CMT

CMT	**F_1 score**	ε
No	0.5	2.56×10^{-2}
Yes	0.67	2.37×10^{-10}

The F_1 score can be interpreted as an evaluation of the relation between the number of false positives, false negatives, true negatives and true positives. It reaches its best value at 1 and its worst value at 0.

We notice that the F_1 score when CMT is used is higher, indicating that a more selective threshold was computed for the cross-validation set. However, the corresponding ε is surprisingly much lower. In terms of detection, we obtain the following results:

- **Without CMT**, the detector reports the attacker VM as being malicious. However, it also reports the victim VM as well as VMs 4 and 5.
- **With CMT**, the detector reports the attacker VM as being malicious. It also reports VMs 5 and 6.

Consequently, we observe that CMT improves the accuracy of our detection approach.

Issues with the Multivariate Model

When using these same features (both with and without CMT) with the multivariate model, we obtain the following error:

```
gsl: cholesky.c:157: ERROR: matrix must be positive definite.
```

This error indicates that the GSL library could not compute the Cholesky decomposition (which is used in the GSL implementation for computing probabilities when using the multivariate Gaussian distribution) of the covariance matrix because it is not positive-definite. There are to possible causes for such errors:

- The features are too related to each other (in other words, not independent enough): in the above example, it might come from LLC references and accesses being two interdependent. However, we have the same error when only using one of these two features, and also when we only use CMT and LLC misses.
- The size of the training set is not much larger than the number of features: in our case, we have 4 features and our training set consists of 6 VMs, which is not enough.

Consequently, with our current setup, we cannot go further with the multivariate model.

Looking for Better Combinations

We made many tests to find a suitable and better combination of features.

We experimented with the following feature, which can be called *LLC miss rate*:

$$\frac{LLC_{misses}}{LLC_{references}}$$

However, it gives unusable values. Indeed, the number of LLC misses is almost always equal to the number of LLC references[3].

As we use our patched *Prime+Probe* attack to target instructions repeatedly used by a program running on a victim VM, we tried leveraging events related to the L1 instruction cache. Therefore, we choose to add the following feature representing read misses in the L1 instruction cache:

$$\frac{L1i_{\text{read misses}}}{ITLB_{\text{read accesses}}}$$

Consequently, we now use the following features in our detector:

$$\text{CMT}, \frac{LLC_{misses}}{ITLB_{\text{read accesses}}}, \frac{L1i_{\text{read misses}}}{ITLB_{\text{read accesses}}}$$

The results are shown in table 4.9.

TABLE 4.9: F_1 score for cross-validation set with and without CMT

CMT	F_1 **score**	ε
No	0.33	6.45×10^{-3}
Yes	0.67	4.64×10^{-11}

Regarding the detection accuracy, we observe the following:

- **Without CMT**, the detector reports the attacker VM as being malicious. However, it also reports all the other VMs except VM 3.
- **With CMT**, the detector reports the attacker VM as being malicious. It also reports VMs 4 and 6.

When CMT is not used, this choice of features is thus less effective than the previous one. However, CMT adds more accuracy to this configuration and the resulting combination yields better results, even if the selected threshold is very low and makes the detection fluctuating.

4.3 Performances

To evaluate the performance of our prototype, we analyze the two following metrics.

4.3.1 Memory Footprint

Using the `htop` command-line tool, we observe that our prototype's process occupies $\approx$ 240 MB of virtual memory and $\approx$ 7 MB of physical memory. This is rather low on the scale of a physical machine hosting VMs. This amount of memory obviously increases with the number of running VMs, the number of monitored events and the size of the sliding window.

4.3.2 CPU Usage

Furthermore, `htop` shows that our process uses between 1 % and 2 % of the CPU and that it basically corresponds to the main thread (which is responsible for probing the `perf` events). This was measured with a probing frequency of 10 ms and the anomaly detection algorithm running every second. For instance, raising the probing

[3]In some cases, it is even higher

frequency to 10 μs increases thread CPU usage up to 50 %. However, as mentioned earlier, such a high probing frequency is not necessary in the case of cache-based side-channel attacks. Regarding the anomaly detection algorithm, the cost is directly related to the number of monitored VMs. We could only run up to 6 VMs with our platform, which can explain the low overhead of our anomaly detection thread.

Finally, because the probing thread needs to run whenever it has to (in order to guarantee the probing frequency), at least one CPU core needs to be *free* (*i.e.* without a VM pinned to it).

Chapter 5

Conclusion and Perspective

In this chapter, we synthesize our work and discuss what could be done in the future to improve our prototype and to extend our approach. Then I summarize what I will retain from this internship.

5.1 Conclusion

During this internship, we focused on the detection of cache-based side-channel attacks. We experimented with a new Intel technology called Cache Monitoring Technology, that allows monitoring the LLC occupancy of a VM. We thus tried to leverage it for detecting such attacks in the cloud environment, in order to react using the MTD approach.

We developed a prototype based on Hardware Performance Counters (HPCs) and CMT, implementing Gaussian anomaly detection. We observed that CMT is interesting to employ for the detection of cache-based side-channel attacks because it provides useful information which is not accessible through other tools/methods (e.g. HPCs).

We also noticed that CMT could (in certain cases and despite its granularity) partially improve the accuracy of HPCs-based anomaly detection techniques. Consequently, we believe that CMT constitutes a promising feature for the detection of cache-based side-channel attacks, especially in virtualized environments such as the cloud.

5.2 Future Work

Implementing a complete MTD-based reaction is the next key development. Thanks to the live-migration techniques implemented in hypervisors such as KVM, migrating a VM to another physical machine is straightforward. However, the MTD approach involves correctly choosing the destination host. This requires accounting and computations to keep everything balanced and to not disrupt operations. Therefore, a separate module will probably be implemented and interconnected as a callback to the detector.

During the evaluation of our prototype, we were quite limited by our attacks. Indeed, we only managed to carry out a portion of one *Prime+Probe* attack. Cache-based side-channel attacks are hard to reproduce and *Prime+Probe* is probably the hardest. For instance, we had relatively more success with *Flush+Flush* attacks but we were not initially targeting such attacks with our detector. Experimenting with more and better *Prime+Probe* attacks would be an interesting continuation of our work to further evaluate and improve our detector.

Furthermore, we did not experiment much with MBM. However, we believe that it could be interesting to leverage this feature to detect cache-based side-channel attacks. For instance, it might be used to indirectly measure LLC misses and thus save one HPC. The probing of MBM `perf` events (local and total) is already implemented in our detector, though the raw value returned by the Linux kernel's Intel RDT support is not a bandwidth but an amount of data (i.e. bytes instead of bytes per second). This brings a few complications and requires a bit more work (the counter keeps increasing thus overflows are to be handled, time flow needs to be accounted, etc.).

Using other anomaly detection techniques is also an interesting option. To that end, it could be easier to use Python's machine learning implementations, such as those included in the `scikit-learn` module. However, interfacing it with our existing C code may be challenging and impact performance.

Finally, the way we need to directly use the Linux kernel API (through the `perf_event_open` system call) and to manually walk `procfs` to retrieve PIDs is not recommended and especially not portable. Exclusively using the `libvirt` API would be cleaner. To this end, submitting patches to `libvirt` in order to bring support for other HPCs could be helpful. The infrastructure is already present since some HPCs (plus CMT and MBM) are already supported by `libvirt`, but the current abstraction layer can be a serious impediment to some use cases. Patches should particularly worry about configurability and flexibility.

5.3 Lessons Learned

In this section, I summarize what I learned during my internship from a technical point of view and I discuss what I discovered from the professional and research worlds. Finally, I reflect on the resulting impact on my professional project.

5.3.1 Technical Takeaways

During this internship, I learned a lot about Intel x86 and ARM microarchitectures. More particularly, I closely studied CPU caches and I also read interesting papers about NUMA[1].

I had the opportunity to practice userspace C programming and to discover the powerful `libvirt` library. I particularly manipulated its API and the setup phase, including the network configuration with `virsh`[2] and QEMU. Finally, the GNU Scientific Library was very interesting to use.

Furthermore, I had the occasion to spend some time looking at the Linux kernel source code. I had been focusing on that during my last school semester but it was interesting to take advantage of those skills in a real context (e.g. when trying to understand the support for Intel cache QoS). Similarly, I rehearsed OS memory management and learned more about KVM and hardware-assisted virtualization in general.

In addition, I was confronted for the first time to machine learning. This was probably the hardest challenge for me during this internship, as I did not have any prior knowledge in this area. I learned about concepts such as supervised and unsupervised learning, anomaly detection, clustering and neural networks. Likewise, I discovered the very inventive MTD approach.

Finally, I learned a lot about how to write a formal report, trying to come as close as possible to the "academic paper" style.

[1] Non-uniform Memory Access

[2] A command-line interface to `libvirt`

5.3.2 Discovery of the Research World

It was my first internship in a big company. It showed me a glimpse of various components of the professional world. For instance, I was curious to observe the functioning of management and the omnipresence of procedures (such as requests for equipments, etc.). I also witnessed for the first time the involvement of trade unions and of the employee representative committee.

I realized that I was in a particular branch of Orange. I was surrounded by research engineers and I guess it contributed to the relaxed atmosphere I had the opportunity to work in. Furthermore, I chose this internship because I was wondering whether I would pursue my studies with a doctoral thesis in order to enter the field of research. I took it as a great opportunity to discover the world of research and see if I would be interested in it. It included how to read academic papers, do a survey, look for ideas, reflect upon a prototype, experiment and interpret measures, evaluate an approach and draw conclusions. I also had many occasions to talk with research engineers and have an insight into their work. This was a valuable experience and I really enjoyed it.

Following those six months, I definitely want to work in an environment where I will face highly technical challenges. However, I think that a more "practical" (or "operational") approach would suit me best. I confirmed that I was particularly interested in systems security and I am looking forward to working in this area.

Appendix A

QEMU-KVM

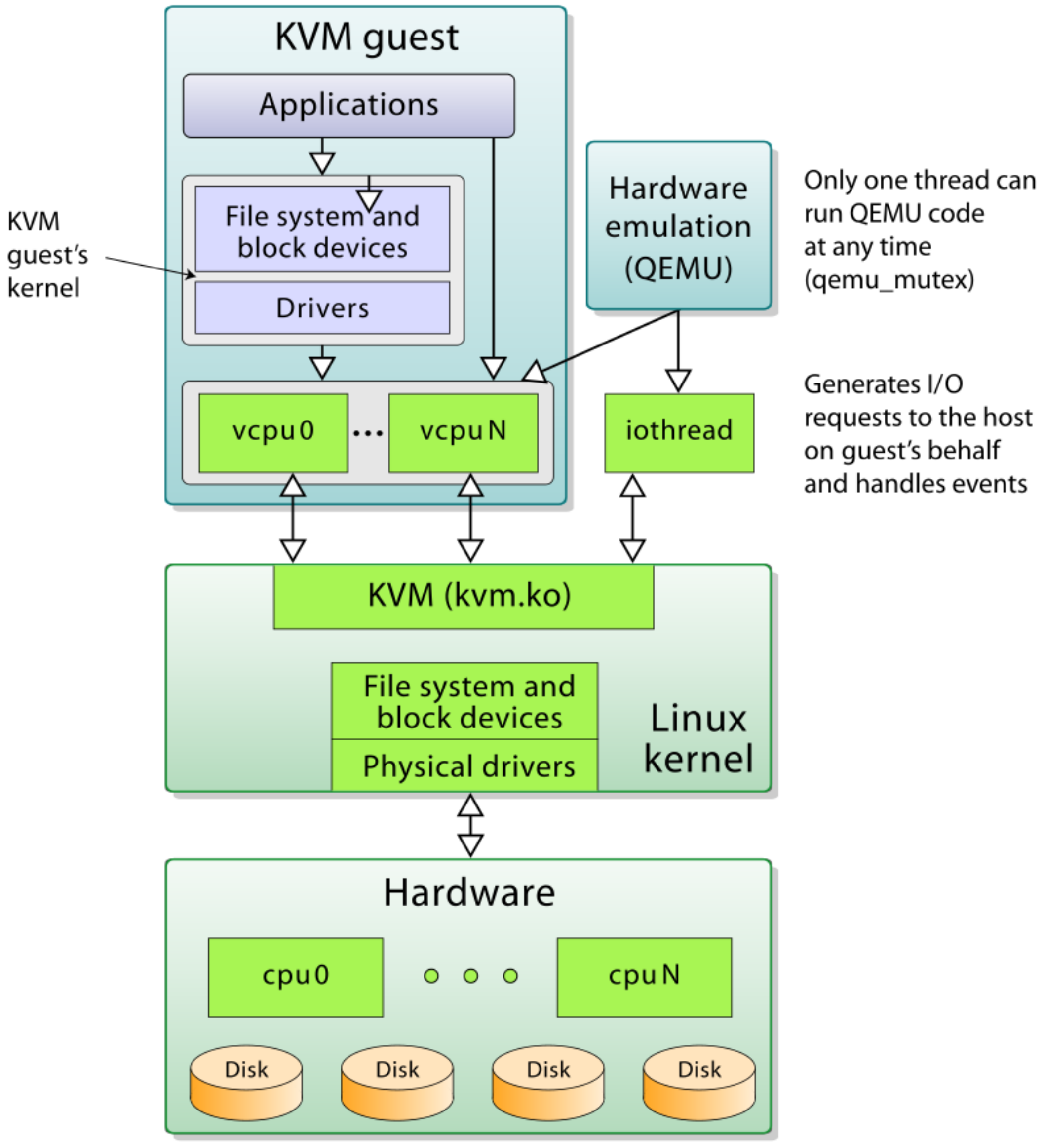

KVM guest
Applications
File system and block devices
Drivers
KVM guest's kernel
vcpu0
vcpuN
Hardware emulation (QEMU)
Only one thread can run QEMU code at any time (qemu_mutex)
iothread
Generates I/O requests to the host on guest's behalf and handles events
KVM (kvm.ko)
File system and block devices
Physical drivers
Linux kernel
Hardware
cpu0
cpuN
Disk
Disk
Disk
Disk

FIGURE A.1: Architecture of QEMU-KVM (Wikipedia)

Bibliography

[1] Onur Aciiçmez. “Yet Another MicroArchitectural Attack:: Exploiting I-Cache”. In: *Proceedings of the 2007 ACM Workshop on Computer Security Architecture.* CSAW ’07. Fairfax, Virginia, USA: ACM, 2007, pp. 11–18. ISBN: 978-1-59593-890-9. DOI: 10.1145/1314466.1314469. URL: http://doi.acm.org/10.1145/1314466.1314469.

[2] Mansaf Alam and Shuchi Sethi. “Covert Channel Detection framework for cloud using distributed machine learning”. In: *CoRR* abs/1504.03539 (2015). URL: http://arxiv.org/abs/1504.03539.

[3] A ARCANGELI. “Increasing memory density by using KSM”. In: *Proc. Linux Symposium, Canada, July 2009.* 2009, pp. 19–28.

[4] Antonio Barresi et al. “CAIN: Silently Breaking ASLR in the Cloud”. In: *9th USENIX Workshop on Offensive Technologies (WOOT 15).* Washington, D.C.: USENIX Association, 2015. URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/barresi.

[5] Mohammad-Mahdi Bazm et al. “State of the Art in Side-Channel Attacks in Cloud Computing”. 2017.

[6] Sergey Blagodurov and Alexandra Fedorova. “User-level scheduling on NUMA multicore systems under Linux”. In: *Linux symposium.* Vol. 2011. 2011.

[7] Daniel G Bobrow et al. “TENEX, a paged time sharing system for the PDP-10”. In: *Communications of the ACM* 15.3 (1972), pp. 135–143.

[8] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. “Real time detection of cache-based side-channel attacks using Hardware Performance Counters”. In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1034.

[9] Jeroen V. Cleemput, Bart Coppens, and Bjorn De Sutter. “Compiler Mitigations for Time Attacks on Modern x86 Processors”. In: *ACM Trans. Archit. Code Optim.* 8.4 (2012), 23:1–23:20. ISSN: 1544-3566. DOI: 10.1145/2086696.2086702. URL: http://doi.acm.org/10.1145/2086696.2086702.

[10] Gustavo Duarte. *Cache: A Place for Concealment and Safekeeping.* http://duartes.org/gustavo/blog/post/intel-cpu-caches/. 2009.

[11] Dario Faggioli. *[Xen-devel] [RFC PATCH 0/7] Intel Cache Monitoring: Current Status and Future Opportunities.* https://lists.xenproject.org/archives/html/xen-devel/2015-04/msg00443.html. 2015.

[12] Anders Fogh. *Cache side channel attacks: CPU Design as a security problem.* https://tinyurl.com/yaza44wh. 2016.

[13] OpenSSL Software Foundation. *OpenSSL - Cryptography and SSL/TLS Toolkit.* https://www.openssl.org/. 2017.

[14] S. Gianvecchio and H. Wang. “An Entropy-Based Approach to Detecting Covert Timing Channels”. In: *IEEE Transactions on Dependable and Secure Computing* 8.6 (2011), pp. 785–797. ISSN: 1545-5971. DOI: 10.1109/TDSC.2010.46.

[15] V. Gopal et al. “Fast and constant-time implementation of modular exponentiation”. In: *28th International Symposium on Reliable Distributed Systems*. Niagara Falls, New York, USA, 2009. URL: http://www.cse.buffalo.edu/srds2009/escs2009_submission_Gopal.pdf.

[16] Daniel Gruss. *IAIK GitHub repository*. https://github.com/IAIK/flush_flush. 2016.

[17] Daniel Gruss, Clémentine Maurice, and Klaus Wagner. “Flush+Flush: A Stealthier Last-Level Cache Attack”. In: *CoRR* abs/1511.04594 (2015). URL: http://arxiv.org/abs/1511.04594.

[18] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches”. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 897–912. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[19] Daniel Gruss et al. “KASLR is Dead: Long Live KASLR”. In: *International Symposium on Engineering Secure Software and Systems*. 2017.

[20] David Gullasch, Endre Bangerter, and Stephan Krenn. “Cache Games – Bringing Access-Based Cache Attacks on AES to Practice”. In: *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. SP ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 490–505. ISBN: 978-0-7695-4402-1. DOI: 10.1109/SP.2011.22. URL: http://dx.doi.org/10.1109/SP.2011.22.

[21] Nishad Herath and Anders Fogh. “These are Not Your Grand Daddy’s CPU Performance Counters”. In: *Blackhat USA 2015*. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf.

[22] *Intel 64 and IA-32 Architectures Software Developper’s Manual*. Volume 3 (3A, 3B, 3C & 3D): System Programming Guide. Intel Corporation. 2016.

[23] G. Irazoqui, T. Eisenbarth, and B. Sunar. “S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES”. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 591–604. DOI: 10.1109/SP.2015.42.

[24] A. Khandual. “Performance Monitoring in Linux KVM Cloud Environment”. In: *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. 2012, pp. 1–6. DOI: 10.1109/CCEM.2012.6354620.

[25] J. Kong et al. “Hardware-software integrated approaches to defend against software cache-based side channel attacks”. In: *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 2009, pp. 393–404. DOI: 10.1109/HPCA.2009.4798277.

[26] Moritz Lipp. “Cache Attacks on ARM”. MA thesis. Graz, University Of Technology, 2016.

[27] F. Liu et al. “CATalyst: Defeating last-level cache side channel attacks in cloud computing”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 406–418. DOI: 10.1109/HPCA.2016.7446082.

[28] F. Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[29] Clémentine Maurice et al. “Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters”. In: *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan,November 2-4, 2015. Proceedings*. Ed. by Herbert Bos, Fabian Monrose, and Gregory Blanc. Cham: Springer International Publishing, 2015, pp. 48–65. ISBN: 978-3-319-26362-5. DOI: 10.1007/978-3-319-26362-5_3. URL: http://dx.doi.org/10.1007/978-3-319-26362-5_3.

[30] Paul E. Mckenney. *Memory Barriers: a Hardware View for Software Hackers*. 2009.

[31] Soo-Jin Moon, Vyas Sekar, and Michael K. Reiter. “Nomad: Mitigating Arbitrary Cloud Side Channels via Provider-Assisted Migration”. In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: ACM, 2015, pp. 1595–1606. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813706. URL: http://doi.acm.org/10.1145/2810103.2813706.

[32] Khang T (Intel) Nguyen. *Intel’s Cache Monitoring Technology: Software Support and Tools*. https://software.intel.com/en-us/blogs/2014/12/11/intels-cache-monitoring-technology-software-support-and-tools. 2017.

[33] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *Proceedings of the 2006 The Cryptographers’ Track at the RSA Conference on Topics in Cryptology*. CT-RSA’06. San Jose, CA: Springer-Verlag, 2006, pp. 1–20. ISBN: 3-540-31033-9, 978-3-540-31033-4. DOI: 10.1007/11605805_1. URL: http://dx.doi.org/10.1007/11605805_1.

[34] Mathias Payer. “HexPADS: A Platform to Detect "Stealth" Attacks”. In: *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639*. ESSoS 2016. London, UK: Springer-Verlag New York, Inc., 2016, pp. 138–154. ISBN: 978-3-319-30805-0. DOI: 10.1007/978-3-319-30806-7_9. URL: http://dx.doi.org/10.1007/978-3-319-30806-7_9.

[35] Chao Peng. *Achieving QoS in Server Virtualization: Intel Platform Shared Resource Monitoring/Control in Xen*. https://tinyurl.com/y95vhens. 2015.

[36] Thomas Ristenpart et al. “Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds”. In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS ’09. Chicago, Illinois, USA: ACM, 2009, pp. 199–212. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653687. URL: http://doi.acm.org/10.1145/1653662.1653687.

[37] Mark Seaborn and Thomas Dullien. “Exploiting the DRAM rowhammer bug to gain kernel privileges”. In: ().

[38] Vikas Shivappa. *Introduction to Cache Quality of service in Linux Kernel*. https://events.linuxfoundation.org/sites/events/files/slides/presentlinuxcon_vikas_0.pdf. 2015.

[39] Konstantin S. Solnushkin. *Harvard Computer Architecture*. 2007.

[40] Raphael Spreitzer and Thomas Plos. “Cache-Access Pattern Attack on Disaligned AES T-Tables”. In: *Constructive Side-Channel Analysis and Secure Design: 4th International Workshop, COSADE 2013, Paris, France, March 6-8, 2013, Revised Selected Papers.* Ed. by Emmanuel Prouff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 200–214. ISBN: 978-3-642-40026-1. DOI: 10.1007/978-3-642-40026-1_13. URL: http://dx.doi.org/10.1007/978-3-642-40026-1_13.

[41] 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.* ISCA ’07. San Diego, California, USA: ACM, 2007, pp. 494–505. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250723. URL: http://doi.acm.org/10.1145/1250662.1250723.

[42] XenProject wiki. *Intel Platform QoS Technologies.* https://wiki.xenproject.org/wiki/Intel_Platform_QoS_Technologies. 2017.

[43] Wikipedia. *Bus snooping.* https://en.wikipedia.org/wiki/Bus_snooping. 2017.

[44] Wikipedia. *Cache coherence.* https://en.wikipedia.org/wiki/Cache_coherence. 2017.

[45] Wikipedia. *Cache replacement policies.* https://en.wikipedia.org/wiki/Cache_replacement_policies. 2017.

[46] Wikipedia. *CPU cache.* https://en.wikipedia.org/wiki/CPU_cache. 2017.

[47] Wikipedia. *Orange S.A.* https://en.wikipedia.org/wiki/Orange_S.A.. 2017.

[48] Jingzheng Wu et al. “C2Detector: A Covert Channel Detection Framework in Cloud Computing”. In: *Sec. and Commun. Netw.* 7.3 (2014), pp. 544–557. ISSN: 1939-0114. DOI: 10.1002/sec.754. URL: http://dx.doi.org/10.1002/sec.754.

[49] xealits. *What are Kernel PMU event-s in perf_events list?* https://unix.stackexchange.com/a/328038. 2016.

[50] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *23rd USENIX Security Symposium (USENIX Security 14).* San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[51] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. “CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds”. In: *RAID.* 2016.

[52] Yinqian Zhang et al. “Cross-VM Side Channels and Their Use to Extract Private Keys”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security.* CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pp. 305–316. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382230. URL: http://doi.acm.org/10.1145/2382196.2382230.

[53] Rui Zhuang, Scott A. DeLoach, and Xinming Ou. “Towards a Theory of Moving Target Defense”. In: *Proceedings of the First ACM Workshop on Moving Target Defense.* MTD ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 31–40. ISBN: 978-1-4503-3150-0. DOI: 10.1145/2663474.2663479. URL: http://doi.acm.org/10.1145/2663474.2663479.