SAPIENZA
UNIVERSITÀ DI ROMA

Detecting Cache-based Side Channel Attacks using Hardware Performance Counters

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Serena Ferracci
ID number 1649134

Thesis Advisor

Prof. Alessandro Pellegrini

Co-Advisor

Dr. Stefano Carnà

Academic Year 2018/2019

Thesis defended on 22/07/2019
in front of a Board of Examiners composed by:

Prof. Fiora Pirri (chairman)
Prof. Aris Anagnostopoulos
Prof. Febo Cincotti
Prof. Pierangelo Di Sanzio
Prof. Umberto Nanni
Prof. Alessandro Pellegrini
Prof. Aurelio Uncini

Detecting Cache-based Side Channel Attacks using Hardware Performance Counters
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: July 15, 2019

Author's email: ferracci.1649134@studenti.uniroma1.it

To you, always in my heart
To my family and friends

Acknowledgments

Vorrei ringraziare la mia famiglia che mi è sempre stata accanto e che mi ha permesso di arrivare fin qui. Perché hanno gioito per tutti i traguardi che ho raggiunto e sono stati la mia roccia nei momenti in cui ne avevo più bisogno.

Vorrei ringraziare tutti i miei amici e compagni di universita, in particolar modo gli amici di sempre: Beatrice, Cristina, Felice e Raffaele. Per esserci sempre nonostante la distanza che a volte ci separata o gli impegni che non ci permettono di essere insieme il tempo che vorremmo. Perché hanno reso questo viaggio pieno di sfide e difficoltà più sopportabile e divertente.

Un particolare ringraziamento va al mio relatore Alessandro Pellegrini e al co-advisor/external-advisor/compagno di banco Stefano Carnà.
A Stefano che ha dovuto sopportare giornalmente tutti i miei problemi e le mie ansie, ma per qualsiasi cosa c'è sempre stato. Per ogni melodrammatico "siamo pronti a riavviare?" e per ogni volta che non funzionava più nulla perché io distrattamente avevo dimenticato di modificare qualche dettaglio. Per le giornate intere passate dentro il dipartimento fino a sera sperando di riuscire a trovare le miracolose metriche. Per tutta la pazienza che ha avuto e per la gentilezza con cui mi ha spiegato più e più volte le cose non mi entravano in testa. Per tutto questo e molto altro ancora, grazie.
Al professor Pellegrini che ha accettato di essere il mio relatore senza sapere che cosa lo aspettasse: le e-mail piene di ansia e le mie continue insicurezze. Per tutti i discorsi di incoraggiamento che ha dovuto fare nel corso di questi mesi. Per tutte le volte che ha cercato di farmi stare "serena". Per aver dovuto correggere ripetutamente questa tesi scritta in un inglese improbabile. Per quel "Lei" che a fatica è riuscito a diventare un "Tu". Ma soprattutto, per tutte le volte che c'era un problema e si è fatto in quattro per risolverlo. Perché, dopo questi 5 anni, non pensavo che avrei trovato un professore cosí disponibile e gentile, quindi grazie mille.

Contents

Chapter 1

Introduction

The advancement of software forced the hardware to become increasingly more efficient. An improvement in performance was achieved thanks to a number of optimizations that have been introduced such as the use of *caches*, *CPU pipelines*, *out-of-order execution* and *speculative execution*. The *caches* are components used by the CPU to reduce the average cost to access data from the main memory, modern CPUs have multiple levels of CPU caches, the Last Level Cache (LLC) is shared between the cores, instead the others are shared only between the process running on the same core . The *CPU pipeline* is a technique used in modern microprocessors which allows to execute a certain number of instructions (depending on the pipeline) in parallel in order to increase the instruction throughput. With the introduction of *out-of-order execution*, instructions are not processed strictly in program order but as soon as all required resources are available. In the end, with *speculative execution*, CPUs use branch predictors to guess which instruction will be executed next, instead of waiting for the result of the previous instructions.

However, the introduction of these optimizations has also exposed systems to new families of attacks, called hardware-based attacks, some of the most famous ones being Spectre, Meltdown and Foreshadow. Hardware-based attacks exploit the previously described optimizations to get information from a victim application. Specifically, the attacker can use the out-of-order execution or the speculative execution to perform operations that in a strictly serialized context would not be allowed, like accessing a memory area for which it has no permissions, or executing (or forcing the victim to execute) a portion of the victim's code that accesses private information. Once the private information is loaded into the cache, the attacker can exploit techniques such as Flush+Reload, Prime+Probe or Evict+Time, to retrieve the information through a side channel.

Software patches have been introduced to try to mitigate these attacks. To prevent cache state effects from spanning across process boundaries, it could be sufficient to disable cache sharing. On a single threaded processor, this requires flushing all caches upon every context switch. On a processor with simultaneous multithreading, it also requires the logical processors to use separate logical caches, statically allocated within the physical cache. Other approaches to cope with this

type of attacks suggest to completely disable the CPU's caching mechanism, or disabling cache sharing only for specific processes (or specific code sections) marked as sensitive. A different approach has been tried with the introduction of a Trusted Execution Environment (TEE) that provides security features such as integrity of applications executing within the TEE, isolated execution, along with confidentiality of their assets. Trusted applications running in a TEE has access to the full power of a device's main processor and memory, while hardware isolation protects these components from user-installed applications running in the main operating system. Software and cryptography-based isolations inside the TEE protect the different contained trusted applications from each other.

The software solutions proposed so far might not be fully suitable to prevent cache-based side-channel attacks. This is because the solutions are tailored to all known attacks they try to prevent, but at the same time more and more advanced attacks are devised, that manage to bypass the introduced software patches. Moreover, these software patches, in addition to not being optimal to limit the exploitability of these attacks, typically add a non-minimal overhead to program execution. This aspect cannot be overlooked, in fact a noticeable drop in performance has been observed when these patches are applied.

In this thesis we explore the viability of a different methodology. Rather than focusing on attack prevention or mitigation, we focus on *detection*. As we have seen, software patches focus on the mitigation and prevention of these families of attacks, while the methodology exposed focuses on detection. The proposal is based on hardware facilities available on modern CPUs, which have been traditionally exploited to profile the behavior of applications from a performance point of view. This technology is based on the availability of Performance Monitoring Counters (PMC), which allow to analyze the behavior of applications at runtime, at a very reduced performance cost. This technology, that is usually used to monitor the entire system or an application, is used to detect at runtime if an application is trying to exploit one or more hardware-based attacks techniques named above. In order to detect a possible attack, we define a set of aggregate metrics, that can be evaluated at runtime, with reduced overhead. Based on the result of these metrics it is determined with a certain degree of confidence if an attack is in progress or not and what is the malicious application involved. If there is a high degree of confidence that an application is executing a malicious code, it can be terminated. To define these metrics used to monitor generic applications, two different approaches were followed. First, we have extensively analyzed the attacks and techniques used to carry out the attacks from a theoretical point of view. Then, both malicious and non malicious applications have been extensively profiled. We also present a preliminary implementation of this methodology based on a Linux Kernel module for Linux operating systems running on x86-64 architectures.

The remainder of this thesis is organized as follows. Chapter 2 provides an overview of optimization techniques introduced in order to speed up executions. The advantages and the disadvantages implicated by the optimization and the families of attacks that exploit them to read and/or write memory of the victim process are also

discussed. Chapter 3 presents the hardware facilities used to detect the described attacks. Chapter 4 explains the methodology used to detect the attack techniques presented and how PMCs can be used for this purpose. Chapter 5 describes a possible implementation of the methodology. Finally, Chapter 6 concludes and sums up this thesis and discusses some possible future works. It discloses how our proposal can be enhanced for an extended support and possible directions for future work.

Chapter 2

Cache-based Side Channel Attacks

Standard computing environments were designed so as to ensure that a process cannot access the address space belonging to another process. Similarly, it is not possible from user mode to access kernel memory. At the same time, a variety of optimization strategies have been introduced over time to improve system performance, which will be explained in detail in the section 2.1, such as caches, pipeline, out-of-order execution and speculative execution. The caches are used to minimize latency to access memory. The pipeline is useful to increase the instruction throughput. The out-of-order execution, instead of enforcing a strict execution order of programs, allows CPUs to reorder instructions. At the end, with speculative execution, CPUs predict the outcome/target of branch instructions. These CPU optimizations have been used to threaten the security guarantees traditionally offered to applications. In particular side-channel attacks (which will be discussed in Section 2.2) allows to bypass traditional memory-protection systems, allowing an attacker to steal information from these applications or the operating system kernel. In general, a side-channel attack is based on gaining information from the implementation of a computer system, instead of weakness in the implemented algorithm itself. The sources exploited to retrieve meaningful information are for example: power consumption, timing information or even sound. The general classes of attacks which are of interest for this thesis are:

- *cache-base attacks*, which are based on the attacker's ability to monitor cache accesses made by the victim in a shared physical system as in a virtualized environment or a cloud service.
- *timing attack*: which are based on measuring how much time various operations (such as reloads or flushes a cache line) take to perform.

2.1 Optimizing Instruction Execution Latency

Instruction latency is the number of processor clocks that an instruction must wait for so that the resources necessary to it are available and therefore released from previous instructions, in the case in which these were in use, or loaded from the

memory. For example, an instruction which has a latency of x clocks will have its data available for another instruction that many clocks after it starts its execution. The factors that have contributed to increasing latency are varied. Those referenced by the optimizations explained in the following section are: access to the main memory for each data that must be used during execution and serialized execution which also forces instructions that are ready to be executed (i.e. all resources necessary for execution are available) to wait because there are instructions before them that are awaiting their resources [27]. To try to mitigate this situation, optimizations have been applied that have brought enormous advantages, but, unfortunately, they have also exposed the system to new families of attacks. The most interesting optimizations for the purpose of this thesis, such as the use of the pipeline and the cache, will be illustrated in the following paragraphs.

2.1.1 Instruction Pipelining

Instruction pipelining is an optimization technique used in the design of modern microprocessors, microcontrollers and CPUs to increase their instruction throughput (the number of instructions that can be executed in a unit of time), parallelizing the elaboration of more than one instruction [26].

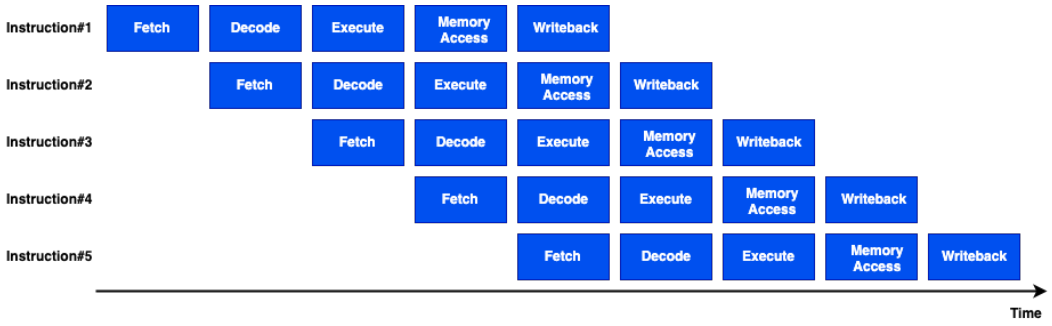


Figure 2.1. Basic five-stage pipeline

A generic pipeline consists of at least 5 stages:

- Fetch: the CPU reads instruction from the memory address stored in the program counter.
- Decode: the encoded instruction present in the instruction register is interpreted by the decoder.
- Execute: ALU operations are performed.
- Memory Access: if necessary, memory operands are read and written from/to the memory.
- Write-back: computed/fetched value is written back to the register present in the instructions.

The number of dependent steps varies with the machine architecture. Modern pipelines have 7, 10 and even 20 stages. As the pipeline is made deeper (with a greater number of dependent steps), a given step can be implemented with simpler circuitry, which may let the processor clock run faster. Such pipelines may be called superpipelines.

If a processor can fetch an instruction on every cycle, it is called to be fully pipelined. Thus, if some instructions or condition require more execution time, causing delays, that prevent fetching new instructions, the processor is not fully pipelined.

Out-of-order Execution and Speculative Execution

Out-of-order execution is a technique that allows to optimize the use of all execution units of a CPU core. Instead of processing instructions strictly in the sequential program order, the execution is parallelized, thus the CPU executes instructions as soon as all required resources are available [18].

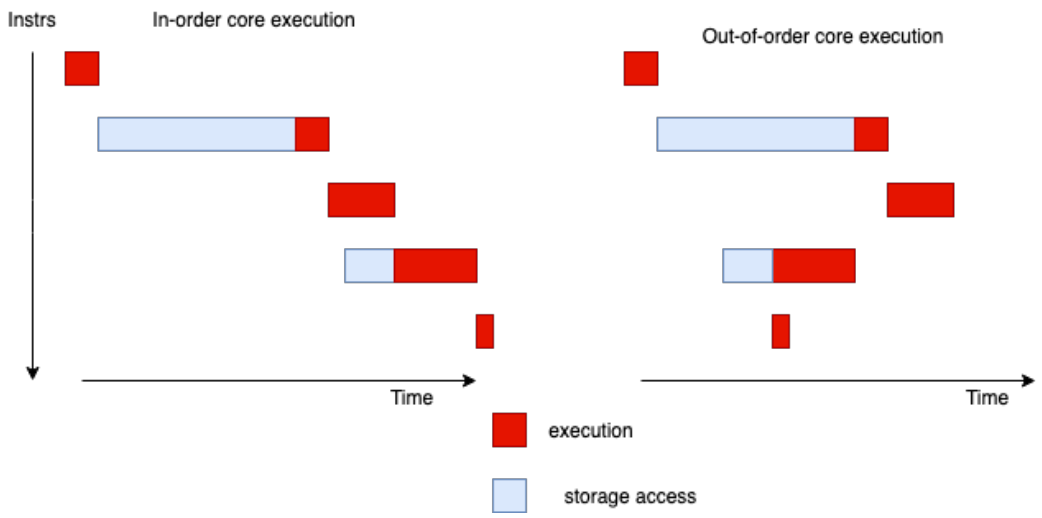


Figure 2.2. In-order vs Out-of-order execution

Due to the fact that CPUs usually do not execute streams of linear instruction, they have branch predictors that are used to obtain an educated guess of which instruction will be executed next, this technique is called *Speculative Execution*. Branch predictors try to determine, before the branch condition is actually evaluated, which is the direction that will be taken or which is the address of the next instruction to be executed before it is retrieved from memory, in case of indirect branch. If the prediction was correct the execution continue. Otherwise, if the prediction was incorrect, a rollback is performed in order to clear the reorder buffer and restart the execution from the correct instruction. The out-of-order execution even if it has no effect on memory, it leaves traces that can be exploited by an attacker.

2.1.2 Optimizing Data Access Latency: The Cache Hierarchy

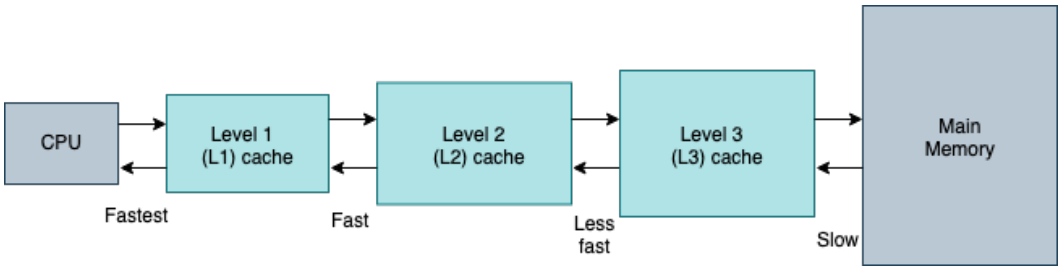


Figure 2.3. Cache Levels

As explained in the previous sections, CPUs are increasingly capable of running and executing larger amounts of instructions in a given time, so the time needed to access data from main memory cannot prevent programs from fully benefiting from this capability. In order to speed-up memory accesses and the Memory Management Unit (MMU), the CPU contains small memory buffers, called caches, that store frequently used data. CPU caches hide slow memory access latencies by storing frequently used data in smaller and faster internal memory buffers. Modern CPUs have multiple levels of caches [5]. The first level is divided in a first level instruction cache and a first-level data cache (L1D). The second level cache (L2) cache is shared by instructions and data. There is an L1 and a L2 in each core. All physical cores are connect to a shared last-level cache (LLC) via a ring connection [11].

When the processor needs to read or write a location in memory, it first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in the cache lines that might contain that address. The check is carried out starting from the first level of the cache. If the processor detects that the memory location is in the cache, then a cache hit has occurred and the access time to that data is small. However, if the processor does not find the memory location in the first level of the cache, it will pass the control to a *cache controller*, that is a hardware block that can automatically move code and data from main memory to the cache memory and back, the controller will look in the second level of the cache, thus continuing to the last level. In case of not even the last level contains the requested information, then a cache error has occurred, the processor must recover the data from the memory and the access will be slow.

2.2 Side Channel Attacks

At a high level, each cache side-channel attack consists of a pre-attack portion, in which important architecture or runtime specific information is gathered; and then an active portion which uses that information to monitor memory accesses of a victim process. The active portion of existing state-of-the-art attacks itself consists of three phases: an *initialization* phase, a *waiting* phase, and a *measurement* phase. The initialization phase prepares the cache in some way; the waiting phase gives the victim process a possibility to access the target address; and then the measurement phase executes a timed operation to establish whether the cache state has changed

in a way that implies an access to the target address has taken place [6].

Specifics of the initialization and measurement phases vary by attack. Some attack implementations decide the length of the waiting phase trying to balance accuracy and resolution, shorter waiting phases give more precise information concerning the timing of victim memory accesses, but may increase the relative overhead of the initialization and measurement phases, which may make it more likely that a victim access could be missed by occurring outside of one of the measured intervals. The attacks generally rely on common techniques extrapolate the secret called: *Flush+Reload* [32][7], *Evict+Time* [8] and *Prime+Probe* [24] [6][19]. The techniques just mentioned are exploited by more elaborated attacks such as *Spectre* [15][21][16], *Meltdown* [17] and *Foreshadow* [31][29]. The most significant difference between them is how the secret is loaded form the memory to the cache.

2.2.1 Spectre

There are three Spectre variants, each of which exploits a different component present in the system. The first two variants take advantage of speculative execution. The first focuses on conditional branches: the attacker mistrains the CPU's branch predictor to mispredict the direction of a branch, causing the CPU to temporarily violate program semantics by executing code that would not have been executed otherwise. Instead, the second focuses on indirect branches: the attacker chooses a gadget, a sequence of machine instructions typically ends in a return instruction located in the memory of a program and/or shared library code, from the victim's address space and influences the victim to speculatively execute the gadget.

```
if(x < array1_size) {
    y = array2[array1[x] * 4096];
}
```

Figure 2.4. Gadget Spectre Attack

The one shown in the Figure 2.4 is an example of the gadget that can be exploited by this attack.

Suppose that the variable x con be controlled by the attackers. In order to prevent a buffer overflow, the gadget contains an *if* statement whose purpose is to verify that the value of x is lower than the maximum array size. The attack, therefore, consists in reading potentially secret data from the process address space. First, during the initial mistraining phase, the attacker invokes the gadget with valid inputs, thus training the branch predictor to expect the *if* condition to be true. Subsequently, during the attacker recalls the gadget with a value of x greater than the maximum size of array1. Instead of waiting for the determination of the result of the branch, the branch predictor guess that the *if* condition is true and that it already executes speculative instructions that evaluate *array2 [array1 [x] * 4096]* using the x passed by the attacker. At this point, *array2* loads the data into the

cache in an address that depends on *array1[x]*. The address is multiplied by *4096* so that accesses go to different cache lines.

Unlike *Return-oriented Programming*[1] (ROP), the attacker does not rely on a vulnerability in the victim code. Instead, the attacker mistrains the Branch Target Buffer (BTB) so that when there is an indirect branch instruction the address returned by the predictor is that of the gadget, resulting in speculative execution of the gadget code. The third variant is slightly different from the previous ones because it uses the Return Stack Buffer (RSB) to perform the attack: the return address value in the RSB is different from the return address value in the software stack, leading the program to misspeculate to the address in the RSB. If this misspeculation can be triggered intentionally by an attacker, he can force a process to execute arbitrary code. Based on how RSB is implemented, the attacker can: overflow or underfill RSB due to limited size, pollute the RSB directly or use RSB across execution context (after a context switch a thread finds in the RSB the addresses of the previous scheduled thread).

2.2.2 Meltdown

It exploits a condition of competition, inherent in the design of many modern CPUs. This occurs between memory access and privilege control during instruction processing. Furthermore, combined with a side cache channel attack, this vulnerability allows a process to ignore normal privilege checks that prevent a malicious process from accessing data belonging to the operating system and other running processes. The vulnerability allows an unauthorized process to read data from any address mapped to the memory space of the current process. Because instruction pipelining is in the processors involved, data from an unauthorized address will almost always be temporarily cached while running out of service, from which data can be recovered. This is possible even if the original read instruction fails raising an exception, as it happens in the example shown in the Figure 2.5

```
uint8_t *probe_array = new uint8_t[256 * 4096];
// Make sure probe_array is not cached
uint8_t kernel_memory = *(uint8_t*)(kernel_address);
uint64_t final_kernel_memory = kernel_memory * 4096;
uint8_t dummy = probe_array[final_kernel_memory];
// handle (eventual) SEGFAULT
// determine which of 256 slots in probe_array is cached
```

Figure 2.5. Example Meltdown Attack

This situation can be managed in two ways. The attacker can fork the attacking application before accessing the invalid memory location that terminates the process, and access only the invalid memory location in the child process. It is also possible to

[1]ROP is an attack that allows an attacker to to induce arbitrary behavior in a vulnerable program, throughput a sequence of machine instructions that are already present in the memory of the victim process.

install a signal handler that will be executed if a certain exception occurs. Otherwise, the attacker can use a different approach to deal with exceptions that is prevent them from being raised. If an exception arise within the transaction, the architectural state is restored and the program execution continues without interferences.

2.2.3 Foreshadow

The vulnerability is a speculative execution attack on Intel processors that may result in the disclosure of sensitive information stored in personal computers and third-party clouds. There are two versions: the first version (Foreshadow) targets data from *Intel Software Guard Extensions* (SGX) enclaves, a private region of memory defined at user-level or operating system code; and the second version (Foreshadow-NG) targets virtual machines (VMs), hypervisors (VMM), operating systems (OS) kernel memory, and System Management Mode[2] (SMM) memory.

The *basic Foreshadow* attack, which can be divided into three distinct phases, extracts a single byte from an SGX enclave. In *Phase I* of the attack, plain text enclave data is cached. Next, in *Phase II* the attacker dereferences the enclave secret and loads a secret-dependent oracle buffer entry into the cache, speculatively executing the transient instruction sequence. Finally, *Phase III* acts as the receiving end of the *Flush+Reload* technique (or one of the other techniques explained below) and reloads the oracle buffer slots to establish the secret byte.

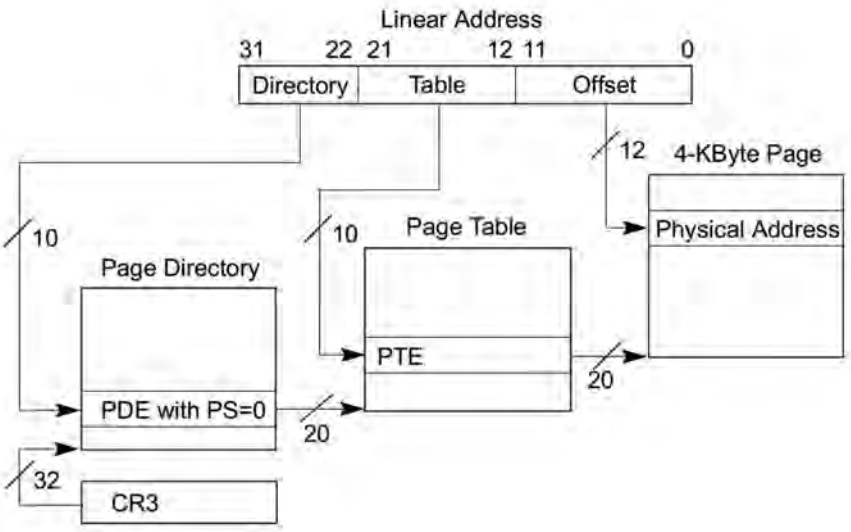


Figure 2.6. i386 Paging Scheme

Foreshadow-OS exploits the features of the paging scheme (shown in Figure 2.6). The scheme is characterized by levels, the level of interest is *Page Table*. Each entry of this level stores a frame number and optional status bits as can be seen in Figure 2.7:

[2]SMM is an operating mode with high privileges that operated outside the view of other software including the OS.

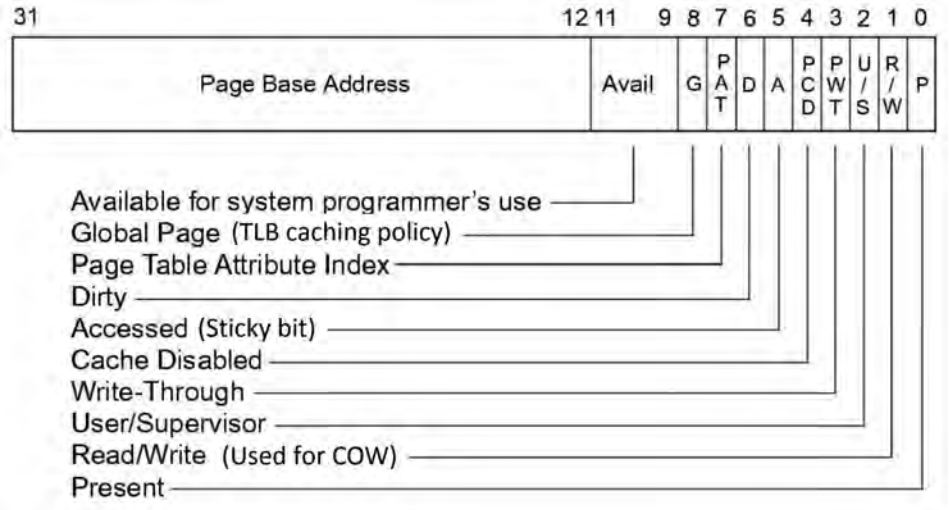


Figure 2.7. Page Table Entry

The bit, which will be exploited by the attack, is the *present* bit. Present bit is set if a particular page we are looking for is present, it is equal to 0 if the page is absent. If we try to access a page that is not present, an error is generated, that is called page fault.

An unprivileged adversary with user space code execution controls the virtual address input to the first page table walk. Such adversaries can cause a page fault, for example by clearing the *present* bit or setting one of the *reserved* bits in a page table entry (PTE). These page faults cause the virtual to physical address translation process to abort immediately and are accordingly referred to as terminal faults (from this the Foreshadow attack is also known as L1 Terminal Fault attack). The terminal faults can be carried out by simply waiting for the OS to clear the PTE present bit in some PTE entry when swapping a page out of memory to disk. At this point, transient out-of-order instructions can be used to read any cached contents located at the physical address pointed by the PTE entry.

Finally, with *Foreshadow-VMM* a malicious guest virtual machine has control over the first address mapping and can thus trigger terminal faults directly by clearing the present bit in the guest page table. Since terminal fault behavior skips the host address translation step and immediately passes the guest physical address to the L1 cache, such adversaries can transiently read any cached physical memory on the system, including memory belonging to other VMs or the hypervisor itself.

2.3 Attack Techniques

After executing one of the attacks shown in the previous section the secret information, which was in the victim's memory area, was cached. The attacker must retrieve the information from the cache so that he can read the secret. This operation is carried out by applying one of the techniques described in the following paragraphs, which exploit the vulnerabilities introduced by the optimizations of the system. The various possibilities, in which an attacker can prepare the attack in order to load

the wanted information in the cache, can be very different between them. Instead these techniques used to retrieve information from the cache all have aspects in common, the most obvious being that all techniques must monitor the cache in one way or another. To do this, as part of the attack preparation, the malicious application should perform two operations. First, it should allocate an *oracle buffer* of 256 slots, each measuring 4 KiB in size (in order to avoid false positives from unintentionally activating the processor's cache line prefetcher). Moreover, the attacker must perform a bunch of tests on the cache by monitoring the access time of a cache line both when the wanted data is in cache and when it must be retrieved from the main memory. At the end of this tests, based on the results obtained, the attacker defines a threshold that will then be used in the course of attack techniques to understand if the cache line monitored was used or not by the victim application. Based on the technique used, the threshold obtained is exploited differently.

- *Flush.* The attacker measures the time to flush a cache line and if the time is greater than the threshold, it can establish that this cache line was used by the victim application.
- *Reload.* The attacker measures the time to reload a cache line and if the time is lower than the threshold, it can establish that this cache line was used by the victim application.
- *Evict+Time.* The attacker measures the time to execute the victim after flush the cache, so if the execution time is greater than the threshold, it can establish that the victim access the cache.

2.3.1 Flush+Reload

To use *Flush+Reload* technique the attacker and the victim need to share the both the cache hierarchy and the memory pages. The technique consists of 3 phases:

- The memory area of interest is flushed from the cache hierarchy.
- The attacker waits to allow the victim to access the cache line.
- The attacker reloads the memory line, while measuring the latency to load it.

The time needed to perform the reload operation is compared to a predetermined threshold. Loads shorter than the threshold are presumed to be served from the cache, indicating that the victim process accesses the memory line after the flush operation. Otherwise, loads longer than the threshold are presumed to be served from the memory, indicating no access to the memory line.

Flush+Flush

A variant of Flush+Reload is the *Flush+Flush* technique. Flush+Flush is a faster and stealthier alternative to existing cache attacks that also has fewer side effects on the cache. In contrast with the technique described above, it does not perform any memory accesses.

The technique consists of only one phase, that is executed in an endless loop. It is the execution of the *cflush* instruction on a chosen shared memory line. The attacker execute the *cflush* instruction and measures its execution time. Based on the execution time, the attacker decides whether the memory line has been cached or not. Since the attacker does not load any memory line into the cache, the execution time reveals whether some other process has loaded it. At the same time, *cflush* evicts the cache lines for the next loop round of the attack.

2.3.2 Evict+Time

The *Evict+Time* technique consists of three steps.

- The victim program is executed and the attacker measures the execution time.
- The attacker evicts[3] a random line in the cache.
- The attacker measures the execution time of the victim again.

The times obtained by the two executions of the same program (one before the eviction and the other after) are compared. If the execution time of the second execution is greater than the first, the cache line was probably accessed during the execution. The attacker needs to predetermined a threshold (computed as described in the in the introduction of the section) to decide if the difference between the two execution times is large enough.

2.3.3 Prime+Probe

The *Prime+Probe* technique is divided into two phases. The attacker, in the *prime* phase, accesses enough cache lines from the cache set to completely fill the cache set with its own data. Later, in the measurement phase (called *probe*), the attacker reloads the same data it accessed previously, this time observing how much time this operation took. If the victim did not access a line of the chosen cache set, the execution time of the reload operation will be lower than the predefined threshold. Instead, if the victim accessed a line of the chosen cache set, a portion of the attacker's data will be no longer cached, causing the execution time of the reload operation to be greater than the predefined threshold because there is a cache miss. Thus, a high execution time implies the victim accessed data in the chosen cache set during the waiting phase. During the *probe* phase, the attacker reloads his data, being this operation equivalent to the *prime* phase, can be used as a *prime* phase for the next cycle of the attack.

Prime+Abort

A variant of Prime+Probe is the *Prime+Abort* technique. This technique can only be used on a specific type of system, namely the transactional memory system. This system tries to simplify parallel programming by grouping read and write operations and executing them as a single operation. Transactional memory is like database

[3]Evict: Access memory until a given address is no longer cached.

transactions in which all accesses to shared memory and their effects are committed or discarded together. All threads can enter the critical section at the same time. If there are conflicts in accessing shared data, the threads try to access the data again or are interrupted without updating the data. Therefore, transactional memory is also called unblocked synchronization. A transactional memory system must preserve the following properties: atomicity, consistency, isolation [10].

In this technique the attacker needs to finds an eviction set (virtual addresses that have the same physical set index). It involves the same *prime* phase as a typical *prime+probe* attack, except that it opens a Transactional Synchronization Extensions[4] (TSX) transaction first. Once the *prime* phase is completed, the attack simply waits for an abort. Since the attacker hold an entire cache set in the write set of his transaction, any access to a different cache line in that set by another process will necessarily evict one of his cache lines and cause his transaction to abort. Thus upon receiving an abort, the attacker can conclude that some other program has accessed an address in the target cache set. For this attack, there is no need to find a timing threshold.

[4]Transactional Synchronization Extensions is an extension to the x86 instruction set architecture (ISA) that adds hardware transactional memory support, speeding up execution of multi-threaded software through lock elision [33].

Chapter 3

Hardware Facilities to Measure Performance

To detect the attacks described above, we propose a methodology which is based on the Intel *Performance Monitoring Units* (PMUs). The most common implementation of these units is represented by Performance Monitor Counters (PMCs), introduced in the Intel Pentium processor. The purpose of the PMCs is to monitor a hardware event, i.e. to gather more or less detailed information on the event. Examples of hardware events are the occurrence of a write operation in memory, a cache miss or the fact that a conditional branch in the execution flow of the program has been taken. The events that can be monitored by this support depend on the available processor architectural family, this is due to the fact that the generation of a hardware event is physically triggered by data paths or control signals implemented in the actual control unit of the CPU, which is often subject to partial or complete re-implementation across different families of processing units. In the literature on hardware profiling, this extremely low-level information is often referred to as micro-architectural events. The benefit of relying on micro-architectural events is that they can be extremely optimized, and they can work at the speed of the CPU. Special registers called model-specific registers (MSR) are used to set and manage PMCs. A MSR is one of several control registers in the x86 instruction set used for debugging, program execution trace and computer performance monitoring. To read and write to these registers the *rdmsr* and *wrmsr* instructions are used respectively. Being privileged instructions, they can only be executed in kernel mode.

To reach the goal of this thesis, this support is used differently, namely used to monitor hardware events that might be proven effective to detect a suspicious behavior in order to determine with a certain probability whether the suspected program is malicious or not. The data useful to understand if a program must be considered suspect or not are extrapolated from the metrics depending on the combined information collected by the PMCs.

3.1 Program Monitoring Counters

There are different types of counters that can be programmed to monitor a different set of events. Those that have been used and that will be analyzed in the following sections are: *Programmable PMC*, *Fixed PMC* and *Offcore PMC*.

3.1.1 Programmable PMC

Programmable PMCs are so defined because it is possible to choose the events to monitor using these PMCs. The configuration and management of these PMCs involve the use of two MSRs: *IA32_PMCx* and *IA32_PERFEVTSELx*. The first is the register that the support uses to count the number of times that the monitored event occurred. The register has a size of 64 bits and once it reaches its maximum value, it starts counting back from 0, moreover it can be written as well as read. The second register, shown in Figure 3.1, is used to manage the PMC.

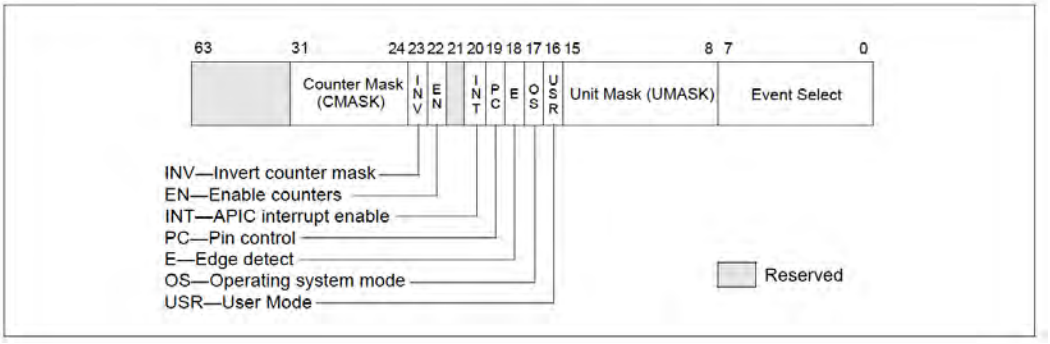


Figure 3.1. Layout of IA32_PERFEVTSELx MSRs

The register fields that can be set in order to decide what and how to monitor are the following:

- *Event Select and Unit Mask.* They are used together to select the event to be monitored. The first one select the event logic unit to monitor an hardware event and the second qualifies the condition that the selected event logic unit detects.

- *User Mode and OS Mode.* If set, they define at which privilege levels (1, 2 and 3 in case user mode is set and 0 in case OS mode is set) the selected event logic unit detects.

- *APIC interrupt enable.* Since the PMCs have a finite dimension, after a certain amount of time they will overflow and start counting again from 0. If this bit is set, the logical processor generates an exception when a PMC overflow.

- *Enable Counters.* If set, the performance count for the selected event is enabled; if it is clear, the corresponding counter is disabled.

3.1.2 Offcore PMC

Uncore is a term used by Intel to describe the functions of a microprocessor that are not in the core, but which must be closely connected to it in order to achieve high performance [9]. In order to monitor these functions, specific events have been introduced, the *uncore events*. Unfortunately, in the processor family used, these events are not available, but have been replaced by the *offcore events* that can be managed through Offcore PMCs. There are two Offcore PMCs that can be activated on each CPU and not four like the Programmable PMCs. Unlike Programmable PMCs, these require programming an extra offcore register that holds filtering information. So to program an Offcore PMC two MSRs must be configured: *IA32_PERFEVTSELx* and *MSR_OFFCORE_RSP_x*. Within the event field and the mask field of the first MRS, one of the pairs of values shown in 3.2 is inserted.

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMC0-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

Figure 3.2. Off-Core Response Event Encoding

These values do not indicate a real event to be monitored, but warn the support that one of the two Offcore PMCs must be used and that the event to be monitored is indicated in the second MSR, *MSR_OFFCORE_RSP_x*. The register is divided into three portion: Bits 15:0 specifies the request type of a transaction request to the uncore, bits 30:16 specifies the response of the uncore subsystem and bits 37:31 specifies snoop response information. To properly program this extra register, at least one request type bit and a valid response type pattern must be set. Otherwise, the event count reported will be zero. It is allowed and useful to set multiple request and response type bits in order to obtain various categories of off-core response events. If the Offcore PMC is activated on IA32_PERFEVTSELx (where x is between 0 and 4), the result is reported in the paired performance monitoring counter (IA32_PMCx MSR).

3.1.3 Fixed PMC

There are some events that are monitored by the system more frequently than others. To avoid having to program PMCs every time there is a need to monitor one of these events, *Fixed-function performance counters* (Fixed PMCs) have been made available. Three Fixed PMCs per thread are available.

Programming the fixed-function performance counters does not involve any of the IA32_PERFEVTSELx MSRs, and does not require specifying any event masks because, as the name suggests, the event monitored by the Fixed PMC cannot be changed. The only register used to program the Fixed PMCs is *MSR_PERF_FIXED_CTR_CTRL* (in Figure 3.3) that provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter:

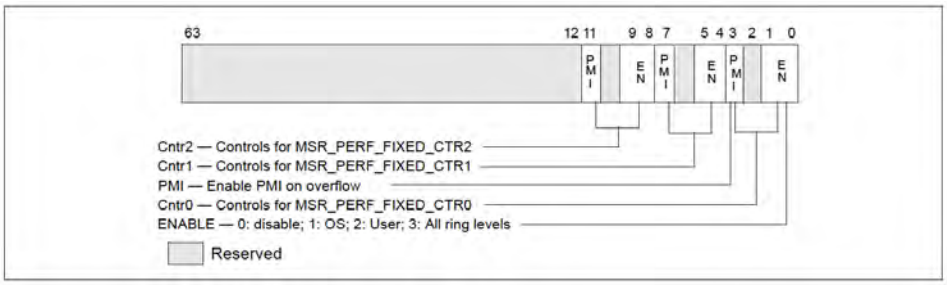


Figure 3.3. Layout of MSR_PERF_FIXED_CTR_CTRL MSR

- *Enable.* When bit 0 is set, the counting is enabled in the corresponding fixed function performance counter to be incremented when the target condition, associated with the architecture performance event, occurs at ring 0. When bit 1 is set, the counting is enabled in the corresponding performance counter of the function to be incremented when the target condition, associated with the architecture performance event, occurs on a ring greater than 0. Writing 0 on both bits, the performance counters stop counting. By setting both the bit 0 and the bit 1, the counter increases independently of the privilege levels.
- *PMI.* If set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

The result of a performance monitoring event is reported in the paired performance monitoring counter the *MSR_PERF_FIXED_CTRx* (where x is the ID of the PMC). Unlike normal PMCs, it is not possible to decide the event to be monitored. Each Fixed PMC has its own event, as shown in Table 3.1:

FIXED_CTR0	INSTR_RETIRED_ANY	This event counts the number of instructions that retire execution.
FIXED_CTR1	CPU_CLK_UNHALTED THREAD_ANY	This event counts the number of core cycles while the logical processor is not in a halt state.
FIXED_CTR2	CPU_CLK_UNHALTED REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state.

Table 3.1. Fixed-Function Performance Counter and Pre-defined Performance Events

3.1.4 Global Registers

In addition to the specific registers for each PMC, global registers were also present. These registers are used to have in a single register an overview of the status of all the PMCs and to be able to manage them more efficiently, for example to turn off or to activate all the PMCs, once configured, it is possible to write only on the

global register and it is not necessary to write to the individual PMC registers. So, to enable a generic PMC, it is not enough to configure the IA32_PERFEVTSELx MSR for Offcore and Programmable PMCs and MSR_PERF_FIXED_CTR_CTRL for Fixed PMCs, it must also be set the corresponding bit in the *IA32_PERF_-GLOBAL_CTRL* register, shown in Figure 3.4.

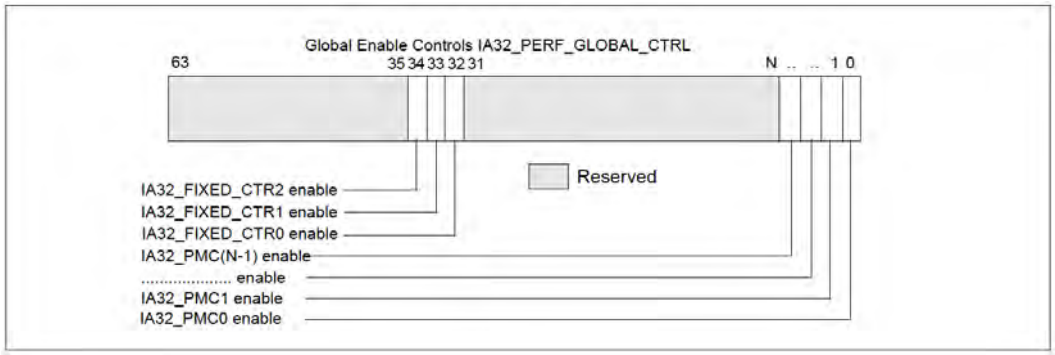


Figure 3.4. Layout of IA32_PERF_GLOBAL_CTRL MSR

For each PMCs that can be activated there is a bit within the MSR. If the bit is equal to 1 and the paired PMC has been configured correctly, then the corresponding monitoring activity will be active, otherwise the PMC will be turned off. This is because the processor performs an AND operation between the enable bit present on the register of the single PMC and the enable bit present in the global register.

In order to manage and to have an overview of the PMCs state, it is possible to use two dedicated registers named: *IA32_PERF_GLOBAL_STATUS*, in Figure 3.5, and *IA32_PERF_GLOBAL_STATUS_RESET*, in Figure 3.6.

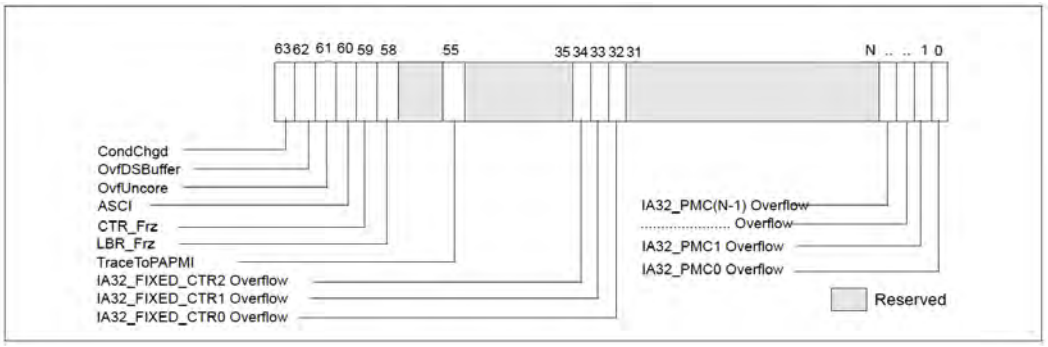


Figure 3.5. Layout of IA32_PERF_GLOBAL_STATUS MSR

IA32_PERF_GLOBAL_STATUS is a read-only register. The first N bits (where N represents the number of available PMCs) represent the state of the corresponding PMC. If the PMCx overflows at least once, the x bits is equal to 1, otherwise it is equal to 0, the registry does not record the overflow number. The same policy is followed for *32nd - 34th bits* that represent the *Fixed Counters*. Another important bit is the *62th bit*: reports if the PEBS records have exceeded the defined threshold.

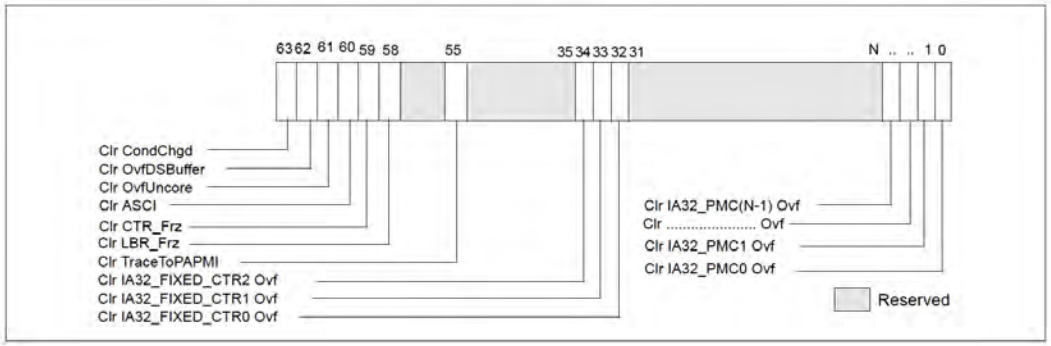


Figure 3.6. Layout of IA32_PERF_GLOBAL_STATUS_RESET MSR

IA32_PERF_GLOBAL_STATUS_RESET is complementary to *IA32_PERF_-GLOBAL_STATUS*. The bits are associated to the same components, but unlike the previous one which is read-only, this registry provides bit fields to clear the *IA32_PERF_GLOBAL_STATUS* indicators.

3.2 Processor Event Based Sampling

A further support of the Intel CPUs, which extends the functionality of the PMCs, is Precise Event-Based Sampling (PEBS) support. It introduces the precise events that can be monitored through PMCs. Unfortunately not all events that are supported by PMCs can be studied using PEBS. PEBS provides a new hardware-based mechanisms that automatically saves the processor context when the counter overflows. This solution, called *PEBS assist*, is implemented at the firmware level, and it avoids any code interruption to gather extra processor information related to the event itselfâĂŤno hardware interrupt is required to save the CPU context, which can be therefore inspected at a later time. PEBS relies on the use of standard meters (PMCs) to function. To activate PEBS on one or more PMCs it is necessary to configure the register shown in Figure 3.7.

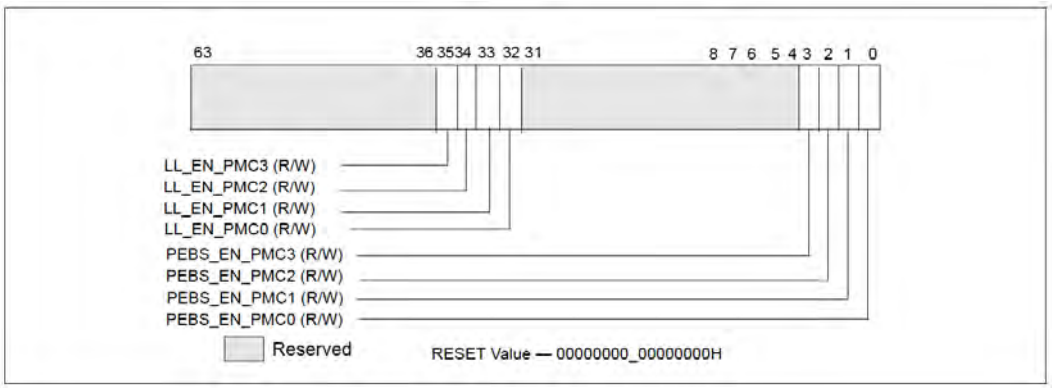


Figure 3.7. Layout of IA32_PEBS_ENABLE MSR

Once one of the PMCs for which PEBS has been configured reaches its maximum value, a hardware interrupt is triggered. At this point, *PEBS assist* is in charge of

collecting information regarding the event that caused the interrupt. The information is packed and saved in a structure called *PEBS record* (shown in Figure 3.8).

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 18.3.6.5.1)
58H	R9	C0H	TSC
60H	R10		

Figure 3.8. PEBS Record Format for 6th Generation Intel CPUs

Some fields of interest of *PEBS record* are:

- *Applicable Counter* indicates which counters triggered the generation of the PEBS record. This allows software to correlate the PEBS record entry properly with the instruction that caused the event, even when multiple counters are configured to generate PEBS records and multiple bits are set in the field.
- *Load/Store Data Linear Address* in case of a load operation, contains the linear address of the source, instead, in case of store operation, contains the linear address of the destination. This field is meaningless in case of other events.
- *Data Source/Store Status* contains three piece of information, as can be seen from Table 3.2:

Field	Bits	Description
Source	3:0	The encoded value indicates the origin of the data obtained by the load instruction
STLB_MISS	4	0: The load did not miss the STLB[5] (hit DTLB or STLB) 1: The load missed the STLB
Lock	5	0: The load was not part of a locked transaction 1: The load was part of a locked transaction.
Reserved	63:6	Reserved

Table 3.2. Layout of Data Source

- *Latency value* contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be zero in that case.
- *Eventing IP* contains the address of the retired instruction that triggered the PEBS assist.

- *TSC* provides a time line annotation for each PEBS record entry.

This record represents the basic element of the *PEBS buffer*. The PEBS buffer is allocated by software and can have a variable size, but should be allocated from a non-paged pool[6], and marked accessed and dirty. This buffer is located in the *Debug store* (DS) area. The DS area presents the structure shown in Figure 3.9.

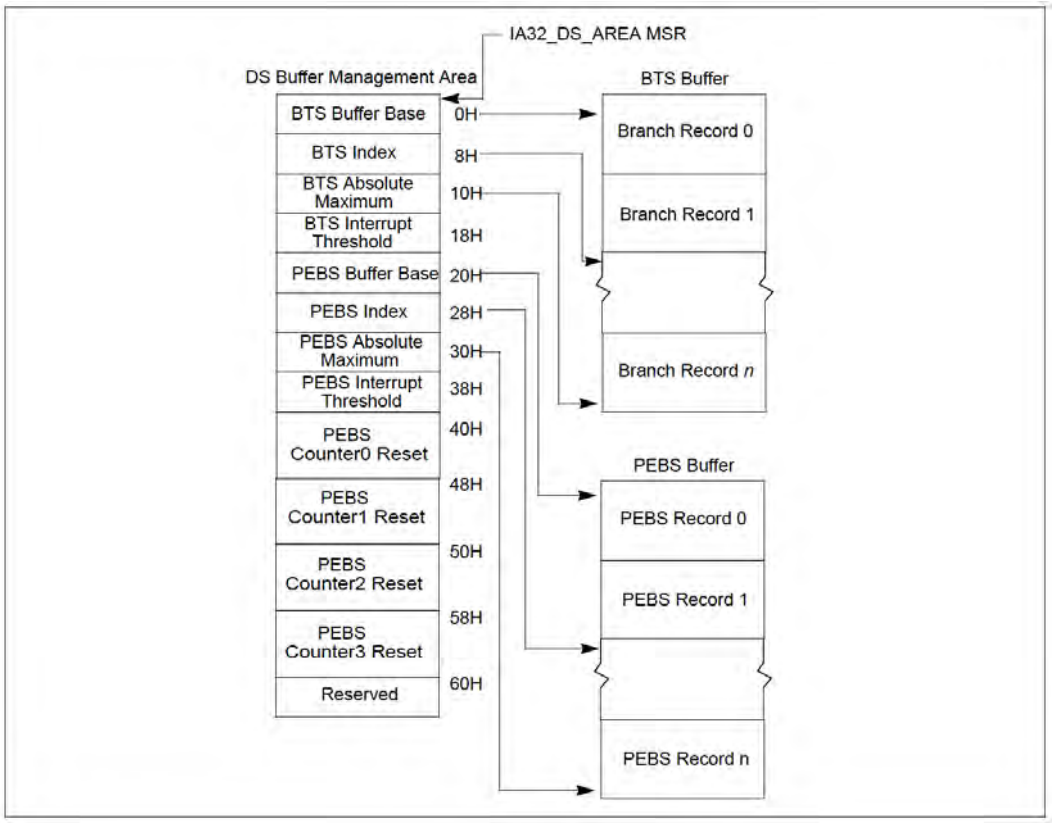


Figure 3.9. PEBS Programming Environment

The *DS Buffer Management Area* is the data structure in memory to support capturing PEBS records for precise events:

- *PEBS Buffer Base*. It is the linear address of the first record of the PEBS buffer.
- *PEBS Index*. At the beginning, this field is equal to PEBS Buffer Base because it identifies the next record to be written when a new interrupt is fired. During the execution, the address is increased by the size of the PEBS record (in this case *C8H*), until the maximum buffer size is reached.
- *PEBS Absolute Maximum*. It is the address of the last PEBS record that can be written. The processor will stop writing PEBS records when PEBS Index equals PEBS Absolute Maximum.

[6]The non-paged pool cannot be paged in and out of physical memory, it is always present.

- *PEBS Interrupt Threshold.* It contains the address of one of the PEBS records. When PEBS Index equals PEBS Interrupt Threshold an interrupt is fired. This interrupt can be distinguished from interrupts caused by PMCs thanks to the use of the register in Figure 3.5, *IA32_PERF_GLOBAL_STATUS*. For the interrupt caused by the PEBS buffer overflow there is a specific bit that is set, 62th bit. Normally, this interrupt is used to notify that the PEBS buffer is nearly full.

- *PEBS CounterX Reset* As mentioned in section 3.1.1, after an overflow the PMC start to count again from 0. Using this field, it is possible to choose the restart value of the corresponding PMC.

Chapter 4

Hardware Performance Counters against Hardware Attacks

The described PMU support is used to monitor the entire system or a single application at runtime. It is usually used through user space programs that allow to select the high-level events to be monitored so as to have information regarding the use of internal resources by single applications. In this chapter, we describe how it is possible to use the same support to detect suspicious behavior within the system. Specifically, the purpose is to understand whether some process is performing a side-channel attack or not.

The PMU support allows to monitor a high number of events. Nevertheless it offers a limited number of PMCs that can be activated on a single core. With the goal to minimize the number of false positives, we have studied the possible events that can be monitored and the attacks, to identify which are the most useful and most significant events for the purpose.

A side-channel attack, as seen in the detailed description of the previous chapters, can be divided into two phases. The first phase is that of preparation, in which the secret information which the attacker wants to obtain must be loaded into the cache. The second phase is that of "extraction", i.e. the information must be read from the cache and moved to an area of memory accessible to the attacker. The latter is realized by exploiting one of the attack techniques described in the Section 2.3. We have studied both phases, so as to identify the events that distinguish a system under attack from a normal execution. To do this, each attack strategy was broken down into operations that correspond to an event that can be monitored. Once this was done, the events that undergo the greatest change were chosen to detect the system as being under attack.

4.1 Side-channel Attacks Study

First a theoretical analysis of the first phase of the attack was performed. To load the cached secret information the attacker can implement different strategies. These

have been studied separately by splitting each of them into simpler operations. Each operation was evaluated and tests were done to see what results could be obtained by exploiting the corresponding events. The most significant operations that emerged from this first analysis are:

- Mistrain the branch predictor
- RSB pollution
- Install a signal handler

However, not all of them are used during monitoring. The attacker needs out-of-order execution or speculative execution in order to load the information in cache. To achieve this purpose, considering the discussed attacks, the attacker can modify directly the RSB or can mistrain the branch predictor. In the first case, the attacker writes directly into the RSB the address he wants it to be executed the next time the return statement is called. instead in the second case, the attacker mistrain the predictor in such a way that if necessary (i.e. every time it has to load the cached secret information), the predictor speculatively chooses the wrong branch. RSB pollution is not a solution adopted often (also because the RSB can also be modified indirectly) and the frequency with which it is written by the attacker is far less than the frequency with which it is written by the processor during normal execution. For these reasons it cannot be used as a metric. On the contrary, since the attacker is forced to read one byte at a time from the victim's memory, it can be concluded that for each byte read the predictor branch must be mistrained, which could produce a significant variation in the monitoring results of the system under attack and during normal execution. Another technique that could be used is to trigger an interrupt and execute the attack inside the interrupt handler, which must have been previously installed. Not even this operation can be monitored for the same reasons as the RSB pollution. The operation is performed only once, while the system during normal execution executes it at a higher frequency. If we tried to monitor RSB pollution and handler installation we would have too many false positives.

4.2 Attack Techniques Study

There are several techniques for retrieving the secret information and moving it to a memory location accessible by the attacker, but they all involve interaction with the cache because that is where the data is located. For this reason in this part of the study we focused on the events that affect the cache, namely:

- Reload cache lines
- Flush cache lines
- Aborted TSX transactions

In most attacking techniques, the attacker accesses the cache, so that the victim application no longer finds the cached data and it is forced to reload it from memory,

and then when it reloads the cache, measuring time of access, will be able to understand if the data that is in the cache line has been loaded by the victim or not. These two operations will cause an increase in the number of cache misses, the level of the cache that will undergo the greatest variation will be identified during the empirical analysis. Another operation that can be performed by the attacker instead of the reload is the *cflush* operation. It is based on the same idea, that the victim is forced to reload the desired data from memory. Once the victim's data load has been executed, the attacker will re-execute a *cflush* operation and based on the time taken by *cflush*, he will understand if the data on the cache line has been loaded or not. *cflush* is different from the reload because the cache is not accessed to perform the operation. For this, the *cflush* must be monitored separately from the reload. The prime + abort technique is based on TSX transactions. The attacker holds an entire cache set in the write set of his transaction and waits until the victim accesses the cache set causing an abort operation. For this reason, also the number of abort operations increases if an attack, that exploits the aforementioned technique, is in progress.

4.3 Attacks and Attack Techniques Profiling

In order to obtain empirical results we distinguished two classes of applications to monitor: non-malicious and malicious applications. We analyzed multiple types of non-malicious applications in order to simulate realistic system load conditions, in particular:

- *FireFox*: while browsing and while playing a video.
- *Gimp*: a cross-platform image editor
- *Document Viewr*: showing and browsing files of different sizes (< 1 MB, $\sim$5 MB and > 25 MB)
- *VLC*, playing a video of 8.12 MB.

We retrieved the source code of the attack techniques described in Section 2.3: *Flush+Reload*, *Flush+Flush*, *Prime+Probe* and *Prime+Abort* [30]. For the sake of this testing phase, in the case of malicious applications, only the attacker is monitored because in a realistic context the detection system does not know which is the target of the attack. The disadvantage of doing this is that we have less information on which infer. On the other hand, this widens the applicability of our approach. These applications have been monitored, in order to evaluate and choose the suitable metrics. The analyzed metrics are both the ones found during the theoretical analysis, to prove or disprove their correctness and its usefulness, and new metrics that during the monitoring could report interesting results for the intended purpose.

Analyzing the events offered by the PMCs to monitor the operations highlighted by the theoretical analysis, we discovered that it is possible to monitor the attack techniques exploited by the attack, but not the attack itself. This is because the attacks are based on out-of-order execution or on speculative execution, while the

events that can be monitored are triggered, in most cases, by *retired instructions*. A *retired instruction* is an instruction that is completely executed, while the attacks, based on the out-of-order execution or speculative execution, execute instructions that would not be allowed in a serialized execution, and consequently these instructions will not be retired, but a rollback will be performed.

Therefore the number of branch mispredictions and the instructions executed in a speculative context cannot be monitored, while it is possible to explore how attack techniques can read the secret information from the cache and this is why we have focused on cache-related events.

4.3.1 Time Slots

Before deciding which events to monitor in order to understand an application's behavior and determine whether the application is performing a cache-based side channel attack or is a non-malicious application, we must decide how often to collect a sample. We need to divide the results into time slots because in this way it is possible to get more information from the analysis of the application. In concrete terms, there is the possibility that a non-malicious application has a flaw and therefore executes malicious code, or a malicious application can have a preamble and a conclusion that mitigate the values assumed by the chosen metrics. In these cases, by studying the aggregated values assumed by the metrics we could have too many false negatives, which is unacceptable. Instead, dividing the execution in time slots it is possible to analyze the execution step by step and better understand the purpose of the application.

To determine a suitableunit of time, the execution time is not used, rather we rely on an event monitored by the PMCs. In particular, the event of interest is monitored by *FIXED_PMC1*, and it is *CPU_CLK_UNHALTED_THREAD_ANY*.

Event	Hex	Description
CPU_CLK_UNHALTED THREAD_ANY	0x30A	Counts the number of core cycles while the core is not in a halt state.

Table 4.1. Time Unit for Sampling

An interrupt is generated each time the *FIXED_PMC1* reaches its maximum value. Inside the interrupt handler the values assumed by the other PMCs are saved and a sample is created. Eventually all the PMCs are restored so that they can start counting again for the new sample. In order for the *FIXED_PMC1* to generate an interrupt at a predetermined time interval, it does not start counting from 0, like the other PMCs. Before starting the monitoring phase, its counter is initialized with a certain value. For example, if we want each 0xFFF (4095 in decimal) events counted by *FIXED_PMC1* to save a sample, the counter associated with *FIXED_PMC1* is initialized to ∼0xFFF (i.e. 0xFFFFFFFFF000) and it is reset to the same value each time an interrupt is generated.

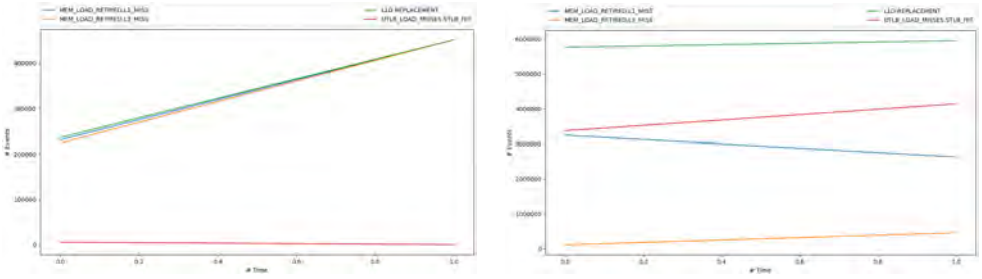
The width of the time slot used during the various tests performed, both to decide which events were monitored and to assess whether, according to the metrics chosen, an application was malicious or not, it was decided by evaluating the implications it

would have on the monitoring results.

In the plots, which will be shown in the following sections, *CPU_CLK_UNHALTED_THREAD_ANY* has been placed on the x-axis. For each time interval, the y values tells the number of times that event occurred.

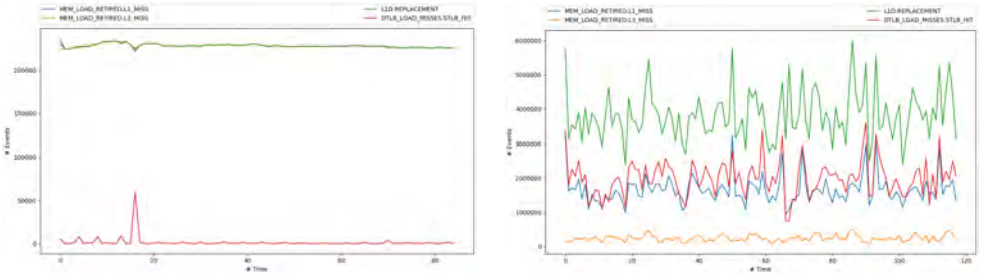
In the case in which the samples are saved with a too low frequency, as in Figure 4.1, we would have results similar to the aggregated values. The values would give too general information on the execution of the application and we could not be establish if the malicious code was injected into the application and if the application, adding superfluous code at the beginning and at the end of the attack, mitigates the metric values. Instead, if the samples are generated too frequently, as in Figure 4.3, the information that a single sample contains is so specific that it does not give us any significant information about the execution of the program. Consequently, the results obtained by a malicious and a non-malicious application might be too similar so as not to allow the distinction of the two.

The frequency chosen to monitor the applications is 7F or 6F, in case the execution of an application is too short and with 7F frequency the samples collected will be too few to allow a precise analysis. An example of applications monitored using this frequency is shown in Figure 4.2, the collected samples provide an exhaustive overview of the application behavior and using this frequency we are not overwhelmed by unimportant information.



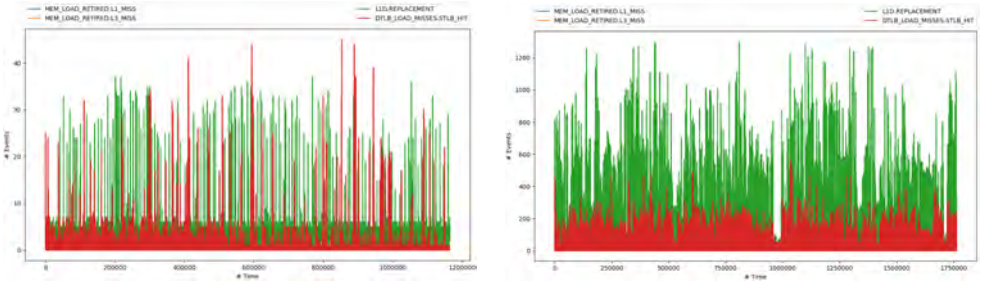
(a) Malicious Application **(b)** Non-Malicious Application

Figure 4.1. Tests performed at 10F Frequency (0xFFFFFFFFFF)



(a) Malicious Application **(b)** Non-Malicious Application

Figure 4.2. Tests performed at 6F Frequency (0xFFFFFF)



(a) Malicious Application **(b)** Non-Malicious Application

Figure 4.3. Tests performed at 3F Frequency (0xFFF)

4.3.2 Cache Events

The theoretical study showed that the area of interest, to study and to detect cache-based side channel attacks, is the use of the cache. As seen in Section 2.1.2, the cache is divided into levels, each with its own parameters. In particular, the hierarchy, used by the microarchitecture of the machine on which the tests were executed, is the one summarized in Table 4.2.

Level	Capacity/ Associativity	Line Size (bytes)	Fastest Latency	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB / 8	64	4 cycles	96 (2*32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB / 8	64	N/A	N/A	N/A	N/A
Second Level	256 KB / 4	64	12 cycles	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44 cycles	32	~18	Writeback

Table 4.2. Cache Configuration of the Skylake Microarchitecture

The cache is divided into three levels, of which only the third level, the last level cache (LLC), is *inclusive*, unlike the others that are *exclusive*. A cache level is exclusive if one of the previous levels can contain an information exclusively, for example the L2 cache contains only the cache lines written-back from L1. Instead, a cache level is inclusive if it includes the contents of all previous levels. An inclusive cache has both advantages and disadvantages. This type of cache is preferable with regards to cache coherence, but containing redundant data (ie already contained in other cache levels) causes a reduction in the global cache size [34].

Given the nature and behavior of the attack techniques we have set ourselves to study, the levels of cache that are studied with particular attitudes are L1 and L3, or LLC. We do not focus on L2 because by monitoring this level we would only obtain information regarding performance. This level could be used to infer information regarding the other two levels, in the event that the support does not make an event available to obtain the desired information. For example, it is possible to obtain the number of misses occurred in L1 through the number of direct requests made to L2, because if a datum is not present in a cache level the data request is propagated to the underlying levels. By studying only the behavior of L1 and L3, we can obtain

information on the number of requests made to the cache and how many of these cause access to the main memory. Through this information we can also get an idea of the location of the application.

After these considerations, we have analyzed and tested the available events, related to the cache, that provide useful information for our purpose.

We have selected the events that best represent the use of caches by malicious and non-malicious applications. The selected events are shown in Table 4.3.

Event	Hex	Description
MEM_LOAD_RETIRED L1_MISS	0x8D1	Counts retired load instructions with at least one uop that missed in the L1 cache.
L1D_REPLACEMENT	0x151	Counts L1D data line replacements including opportunistic replacements, and replacements that require stall-for-replace or block-for-replace.
MEM_LOAD_RETIRED L3_MISS	0x20D1	Counts retired load instructions with at least one uop that missed in the L3 cache.

Table 4.3. Monitored Events

Hit

The information provided by hits and misses are complementary, because for each request only two outcomes are possible: hit or miss. However, we decided to study the number of miss operations because it was the most significant of the two, considering the operation of attack techniques, removing data from the cache and forcing running applications (including the attacker) to retrieve data from main memory, and considering the purpose of this study, to understand how aggressive an application is to memory and therefore the impossibility of exploiting the principle of locality.

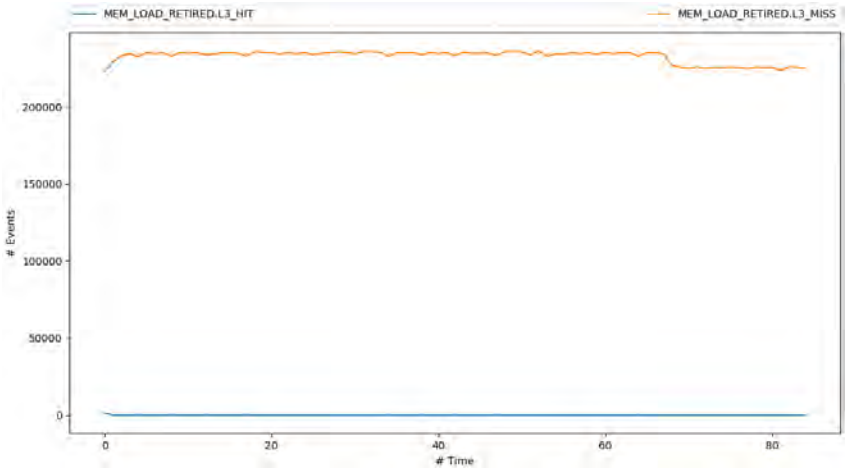


Figure 4.4. Number of ■ L3 Misses compared with number of ■ L3 Hits

Furthermore, as shown Figure 4.4, the oscillations caused by misses are greater than those caused by hits, thus favoring comparison and distinction between malicious and non-malicious applications.

The same consideration was made for L1 and in fact from Figure 4.5 it is possible to notice that similar results are obtained.

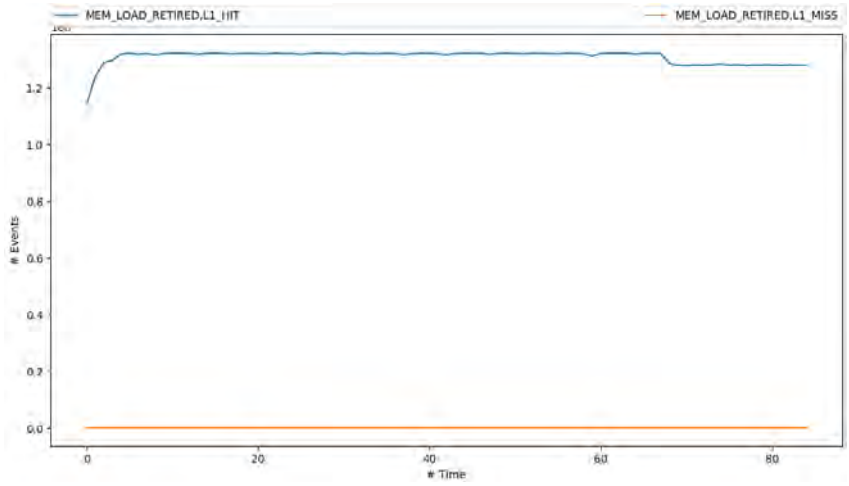


Figure 4.5. Number of ■ L1 Misses compared with number of ■ L1 Hits

Store and Load

The attack techniques focuses on reloading cached data several times to understand if it was already present in the cache or not and and trying to steal information owned by the victim application. Consequently, the operation that characterizes this typology of techniques is precisely the load operation. Going to study the frequency of this operation and the outcome, if it can find the data in cache or if this must be recovered from the main memory, it is possible to make a distinction between the various types of applications. The types we will focus on are those that perform a large number of loads that cause access to main memory, such as malicious applications but not only them, and those that are able to exploit the principle of locality and then recover the data searched by the cache. It follows that the number of stores made by these attacks is so limited that it does not create an obvious fluctuation in the recorded data. This is why we decided to focus on the load operation rather than the store operation.

Furthermore, the support used does not provide as many events with which to monitor store operations in relation to the cache as those to monitor the loads.

I-cache

As Figure 4.2 shows, the first cache level is divided into the sections: data (*D-cache*) and instruction (*I-cache*) sections. While the *D-cache* is accessed only for loads and stores, the *I-cache* is accessed for each instruction. Using the *I-cache*, the events return information related to the size and the structure of the executable. This information is not relevant because the footprint of the executable is not a feature

that can be used to distinguish malicious from non-malicious applications, since the attack techniques that interest us target data and not instructions. Usually, these types of attacks have a reduced structure since they do the same procedure several times, but may also exist non-malicious applications which have the same reduced structure because their operation requires it. Furthermore, the attack structure may vary because a preamble and/or a conclusion may be added, which may be necessary for the execution or to make the malicious application more similar to a non-malicious one.

L1D Replacement

When a cache row is brought into the L1 cache, if the associated cache line is already filled, the cached row must be evicted to make room for it. When the lines in active use are evicted, a performance problem can derive from the continuous return of the data in the cache. This event measures the number of rows replaced into the L1D, i.e. the *D-cache* [4]. This event is monitored, for most of the attack techniques studied invalidate one or more cache lines to be able to retrieve the secret information from the victim application. This situation causes a decrease in the number of rows replaced during application execution.

4.3.3 TLB Events

At this point we have all the interesting information regarding the use of caches, but we still do not have a complete overview on the behavior of the application in order to understand if the monitored application is malicious or not. Another useful aspect to be analyzed is how the *Translation Lookaside Buffer* is used by this application.

The *Translation Lookaside Buffer* (TLB) is a buffer that the Memory Management Unit (MMU) uses to speed up the translation of Virtual Addresses. The TLB has a fixed number of Page Table elements and it is used to map *Virtual Addresses* to *Physical Addresses*. The virtual memory is the space seen by a process and it can be larger than physical memory. This space is catalogued in pages of predefined size. Generally only some pages are loaded into physical memory in areas dependent on the *Page Replacement* policy. The Page Table is used to keep track of where virtual pages are loaded into physical memory. The TLB is a cache of the virtual to physical address translation, i.e. only a subset of its content is stored [22].

To further improve performance, Intel implemented a split TLB architecture, as shown in Figure 4.6, which separates the cache into two disjoint sets. The iTLB handles translations for instruction fetches and the dTLB handles translations for data fetches. In newer CPUs, Intel added a secondary cache called the STLB (*Second Level Translation Lookaside Buffer*), which stores the evicted entries from the iTLB or dTLB.

The TLB provides us with information on the memory area used by an application. Looking at a system in normal load conditions, if an application performs more TLB misses than another application we can deduce that the first application will use a larger memory area than the first. The misses are due to the loading of new pages in the TLB or to the replacement of pages present in TLB, because there is no more space to load a new one. Since the TLB is in common with other processes,

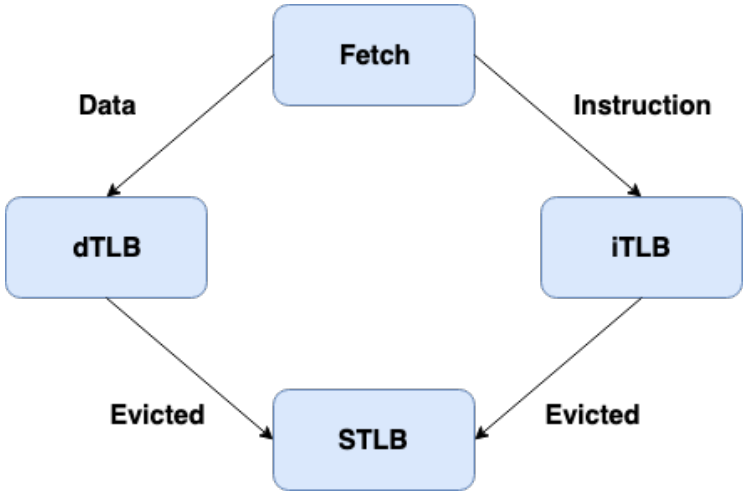


Figure 4.6. TLB Architecture

the replacement may be due to a saturation of the TLB caused by applications other than the monitored one. During our analysis phase, this problem has been investigated by trying to keep the system at a stable workload so that the disturbance due to other applications was reduced to the minimum possible and repeating the tests several times to confirm the results obtained.

Once the usefulness of the TLB was confirmed, we had to choose which operation performed by the TLB would have been more appropriate for monitoring. The PMCs support the monitoring of various types of events related to the TLB and therefore we performed various tests in order to understand which event was the most appropriate. After these tests, the chosen event is *DTLB_LOAD_MISSES_STLB_HIT* and in the subsequent subsections the texts and the reasoning carried out that led to this decision will be exposed.

Event	Hex	Description
DTLB_LOAD_MISSES STLB_HIT	0x2008	Counts loads that miss DTLB (Data TLB) and hit STLB (Second level TLB).

STLB

As described in the previous section, the TLB is divided into two levels. The first decision to be made was which of the two levels to study to get more information on the applications analyzed.

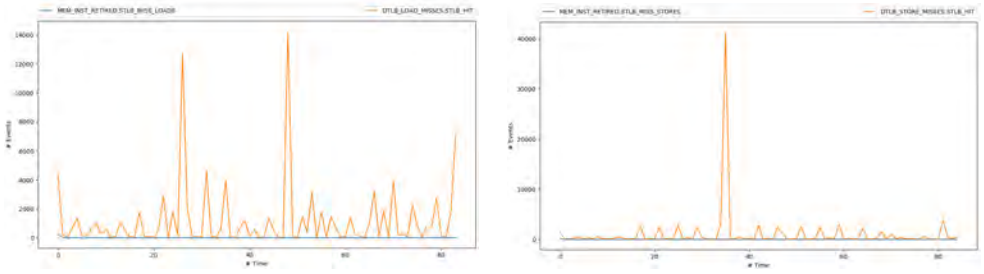
As we can see from Figure 4.7, the first level of the TLB has 128 entries for the instructions and 64 entries for the data, while it has 1536 entries for the second level of the TLB.

In addition to having a fairly high number of entries shared by instructions and data, the STLB also provides a prefetching mechanism that tries to minimize the number of misses, trying to detect strided behavior or relying on the past behavior of the application, to improve the performance [25].

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

Figure 4.7. TLB Parameters of the Skylake Microarchitecture

On the basis of this we can deduce that the samples obtained from events related to the STLB do not contain information relevant to the analysis we are doing and, moreover, could be altered due to the optimizations applied. In support of the deduction we made, we conducted some tests, monitoring some malicious and non-malicious applications, to examine the information that could be obtained from events relating to the STLB, which were compared with those that could be obtained by monitoring the first level of the TLB.



(a) Load Operation **(b)** Store Operation

Figure 4.8. ■ STLB Event compared with ■ dTLB Event

From the tests carried out and shown in Figure 4.8, it is possible to observe that the oscillations caused by the events concerning the *STLB* are lower than those caused by the events on the first level of the TLB. We can deduce that the area of memory used by the application, malicious or non-malicious, is quite extensive in order to cause a miss on the second level of the TLB. So we excluded the events that had *STLB* as their subject because they did not provide enough information to understand the behavior of the application.

iTLB

Once we have established that we are interested in the first TLB level we must decide whether to use the data or instruction events, since the first level is divided into two.

Some previous studies, like the one proposed in the *Flush+Flush: A Fast and Stealthy Cache Attack* paper [7], use *iTLB* events (in particular the number of misses) to gather information on the progress of the application. When, using the *iTLB*,

the events return information on the size of the executable because a new virtual to physical translation is loaded in the *iTLB* if the translation is not already present in the TLB. Thus, we would be able to distinguish an application with a lot of instructions and one with few instruction through the oscillation of samples values. Therefore these studies imply that there are not non-malicious applications with few instructions and malicious applications with a lot of instructions. For this reason, we excluded the events on *iTLB*.

dTLB

The *dTLB* event provides information on the amount of memory accessed by the applications. If the number of TLB misses is low we can say that the application is accessing a limited memory area and there is no need to cache new elements of the Page Table in the TLB. Conversely, if there is a large number of TLB misses, the application is accessing a large memory area and the elements of the Page Table cached in the TLB are replaced frequently. It could be needed to reload an element of the Page Table also if other applications, different from the one monitored, saturate the TLB or if the TLB is explicitly flushed. Both these problems were analyzed to see if they could cause a disturbance such as to distort the results.

- *Noise*: The tests were run trying to keep the system at a constant workload so that even the number of replacements due to the execution of unmonitored applications remained constant. Furthermore, the tests were run more than once to limit the error.
- *Direct flushes*: During monitoring phase, the system could perform a direct flush of the TLB thus causing a noticeable increase in misses within the TLB. To understand if there were situations of this type, we monitored the events described in Table 4.4 which give us information on the number of direct flushes performed during monitoring.

Event	Hex	Description
TLB_FLUSH_DTLB THREAD	0x1BD	Counts the number of DTLB flush attempts of the thread-specific entries.
ITLB_ITLB_FLUSH	0x1AE	Counts the number of flushes of the big or small ITLB pages. Counting include both TLB Flush and TLB Set Clear.

Table 4.4. Direct Flush Events

The results, shown in Figure 4.9, indicate that the number of direct TLB flushes is so low as not to cause distortions in the results obtained by the other events.

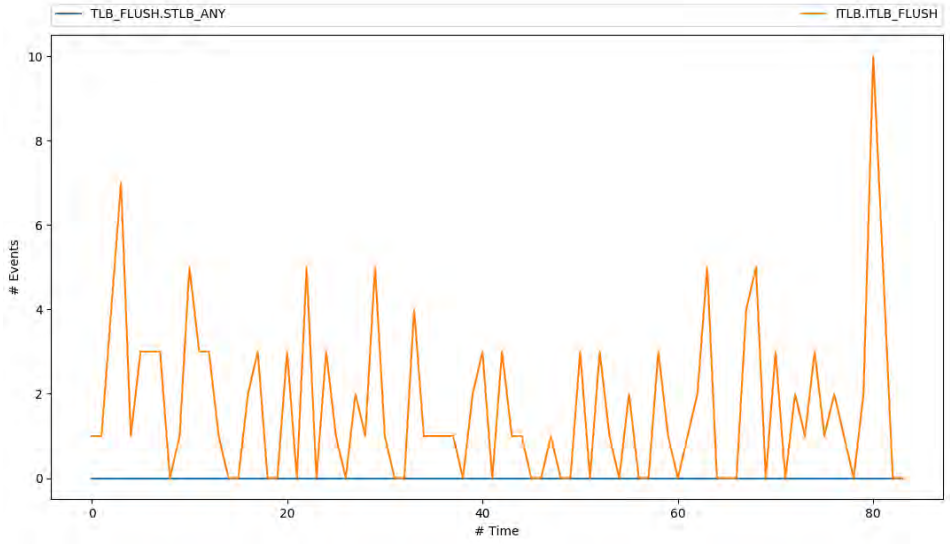


Figure 4.9. Number of Flushes in iTLB and dTLB

In addition, all the tests shown were conducted both on the load and on the store operation. Both operations found that dTLB offers more information to understand the behavior of an application. However, as far as the dTLB event is concerned, the data reported in Figure 4.10 showed that the load operations were causing a greater oscillation and therefore showed more the difference between a malicious and a non-malicious application.

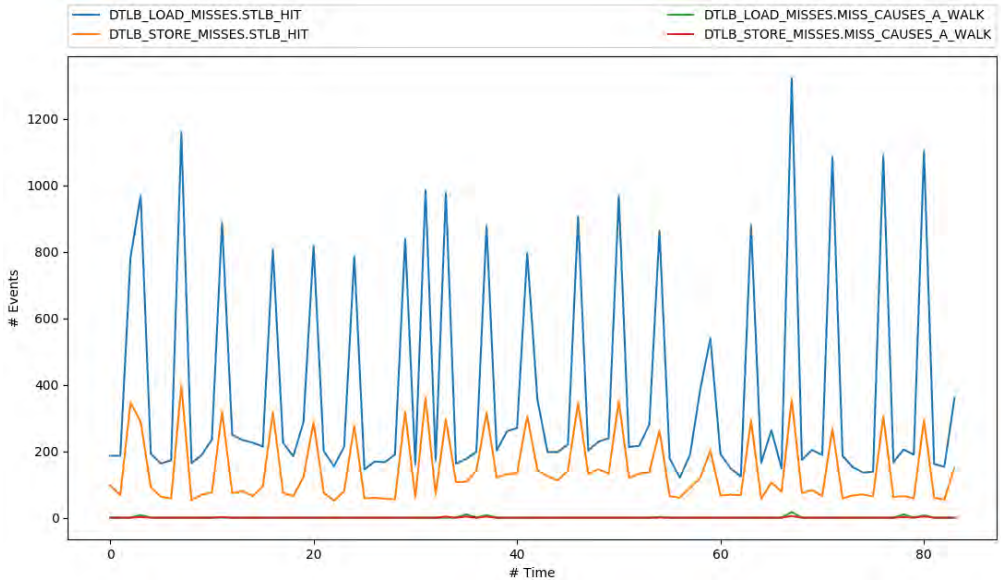


Figure 4.10. Load operation compared to Store operation

This is also confirmed by the fact that the attacks and attack techniques we are studying are based more on reading than writing. For example, each time the reload operation is performed, the application reads a cache line. Once the operation of

interest was identified, we analyzed the events offered by PMC. The events that we could monitor are three and are showed in Table 4.5:

Event	Hex	Description
DTLB_LOAD_MISS CAUSES_A_WALK	0x108	Counts demand data loads that caused a page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels, but the walk need not have completed.
DTLB_LOAD_MISS WALK_COMPLETED	0xE08	Counts demand data loads that caused a completed page walk of any page size (4K/2M/ 4M/1G). This implies it missed in all TLB levels. The page walk can end with or without a fault.
DTLB_LOAD_MISSES STLB_HIT	0x2008	Counts loads that miss DTLB (Data TLB) and hit STLB (Second level TLB).

Table 4.5. DTLB Events

As shown in Figure 4.11, the event that gives us a more complete overview of the program behavior is DTLB_LOAD_MISSES_STLB_HIT because it is the one that undergoes the greatest fluctuations. The other events, on the other hand, have a frequency too low to allow us to make inferences about the results obtained, or they are events too rare to be studied.

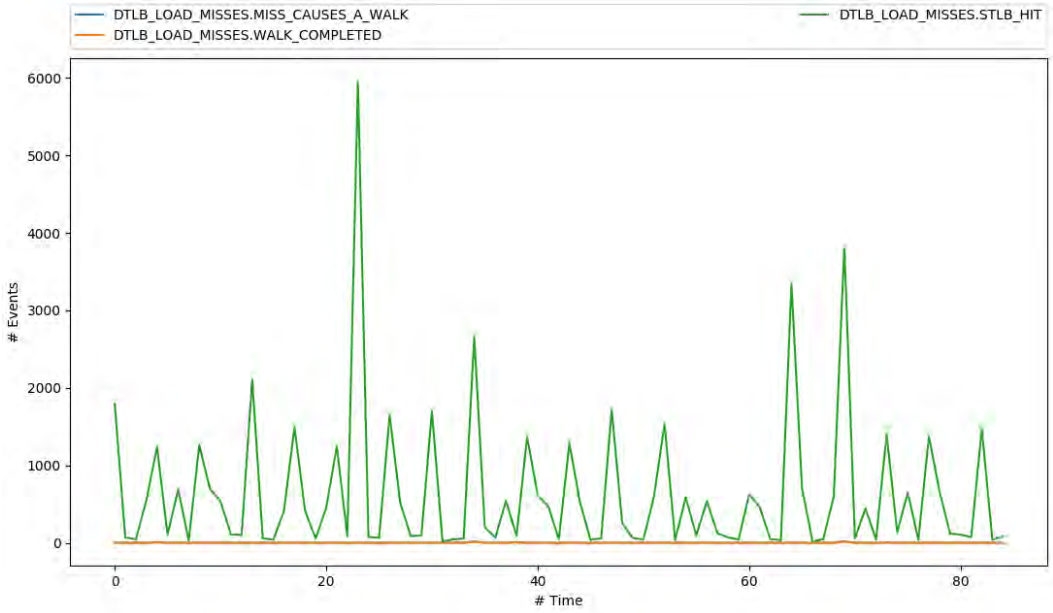


Figure 4.11. Possible monitorable events related to dTLB load

4.4 Metrics

These events, taken individually, cannot be effectively compared because depending on the application, they can provide different information about its behavior; for example, a large number of L3 misses can be both an identification of a malicious application or of un application accessin a large amount of data stored non locally in memory. Therefore, aggregated metrics have been determined in order to obtain meaningful results for the analysis we want to perform.

The established metrics are the following, which we shall discuss later in this section.

1. $\frac{\texttt{MEM\_LOAD\_RETIRED\_L3\_MISS}}{\texttt{L1D\_REPLACEMENT}}$ 2. $\frac{\texttt{DTLB\_LOAD\_MISSES\_STLB\_HITS}}{\texttt{MEM\_LOAD\_RETIRED\_L1\_MISS}}$

4.4.1 Cache Locality

In the first metric, L3 miss indicates the number of times accessed data was not found in the cache, that causing a load from memory. Consequently, the data has been replaced with other data because the monitored application and other applications concurrently running in the system access other data in memory, or the cache has been invalidated and the application is forced to access main memory. L1D Replacement indicates how many times the lines in the first level of the cache have been replaced. In this way we try to differentiate the two scenarios described above. The behavior of the application that we want to capture with this metric is a remarkable use of main memory associated with a small number of replacements in the first level of cache, which could indicate that the cache has been invalidated.

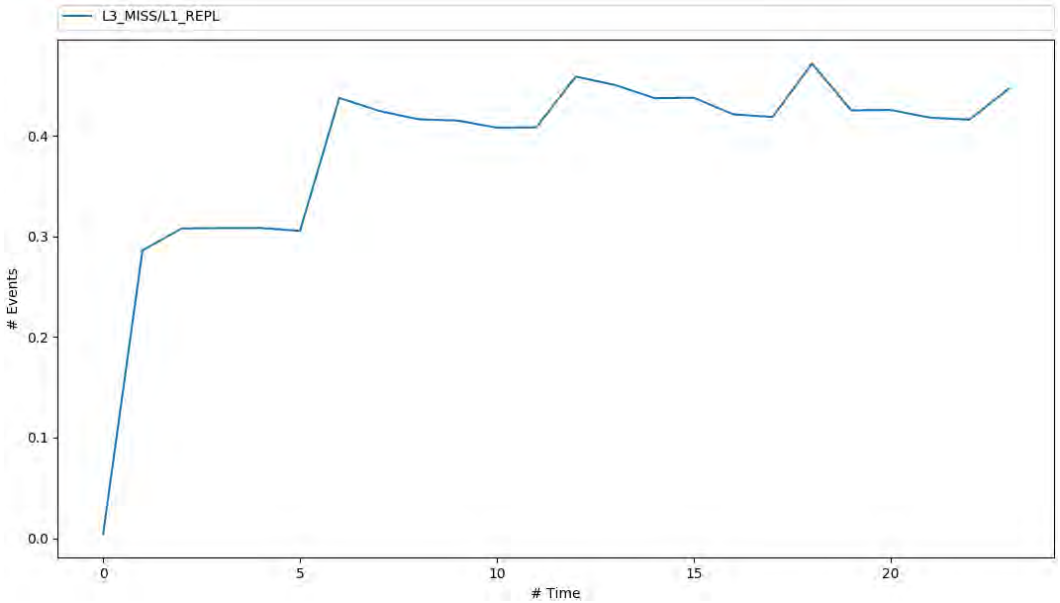


Figure 4.12. Malicious Application (Prime + Probe)

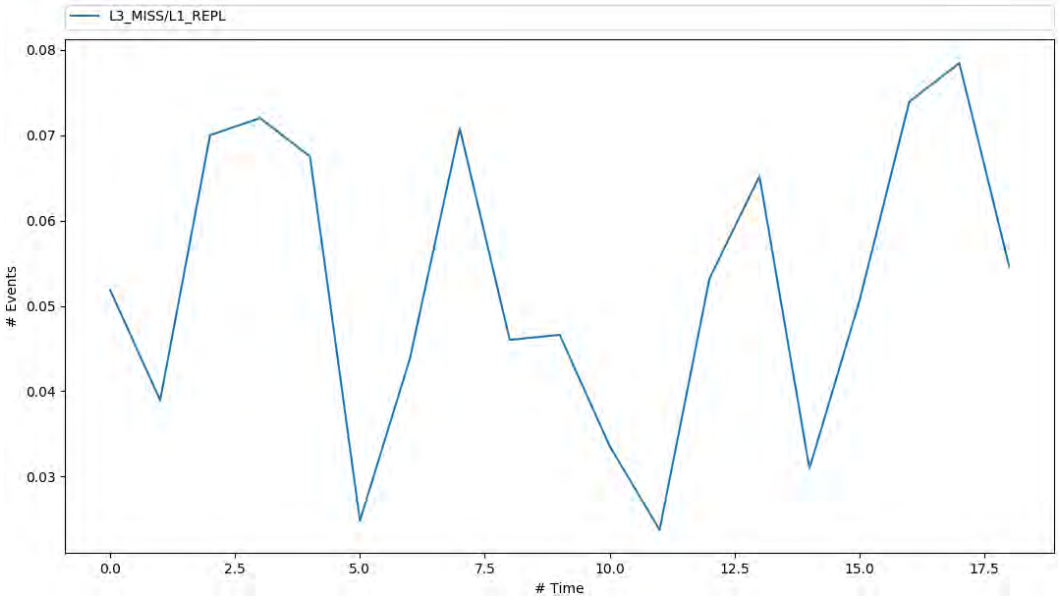


Figure 4.13. Non-Malicious Application (Gimp)

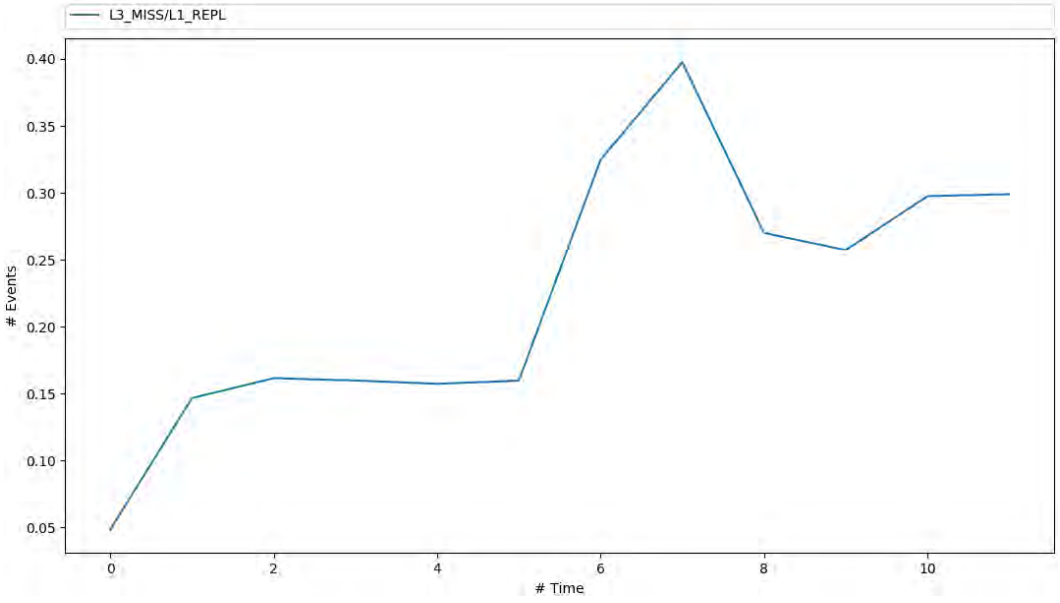
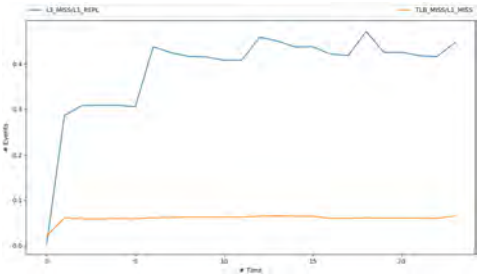


Figure 4.14. Non-Malicious Application: False Positive (Canneal)

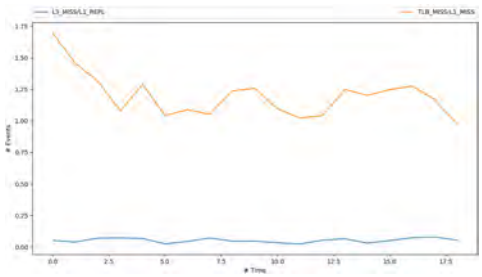
Thus, a malicious application is identified by a high value of this metric. Unfortunately, only this metric is not enough to distinguish malicious applications from non-malicious ones. Indeed, as shown in Figure 4.14, even a non-malicious application can reach a high value and have a trend similar to that of a malicious application. We can conclude that the metric represents a signal that the application may be malicious, but we need an additional metric to minimize false positives.

4.4.2 Cache - Working Set Relation

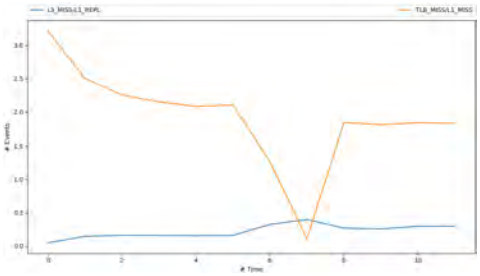
In the second metric introduced, L1 miss indicates the number of times accessed data was not found in the first level of the cache and DTLB miss denotes how many times the virtual to physical translation of the memory address was not found in the first level of the TLB. So, if the value of this metric is greater than 1 we can guess that, given the number of cache misses on the first level of the cache, the number of misses on the TLB, and therefore the area of memory used, is greater. As a result, the cache is used appropriately and the cache misses are due to an aggressive use of memory. Conversely, if the metric has a value between 0 and 1, we have that the number of misses on the first level of cache is greater than the number of misses on the TLB. So we can assume that even if the application working set is limited (therefore being able to exploit the principle of locality) the number of cache misses is however high. The extreme case is obtained when the value approaches 0 and therefore the misses on the TLB are rare because the application is concentrating the access on a restricted area of memory, while the application is continuously forced to retrieve data from lower cache levels or from the main memory.



(a) Malicious Application (Prime+Probe)



(b) Non-Malicious Application (Gimp)



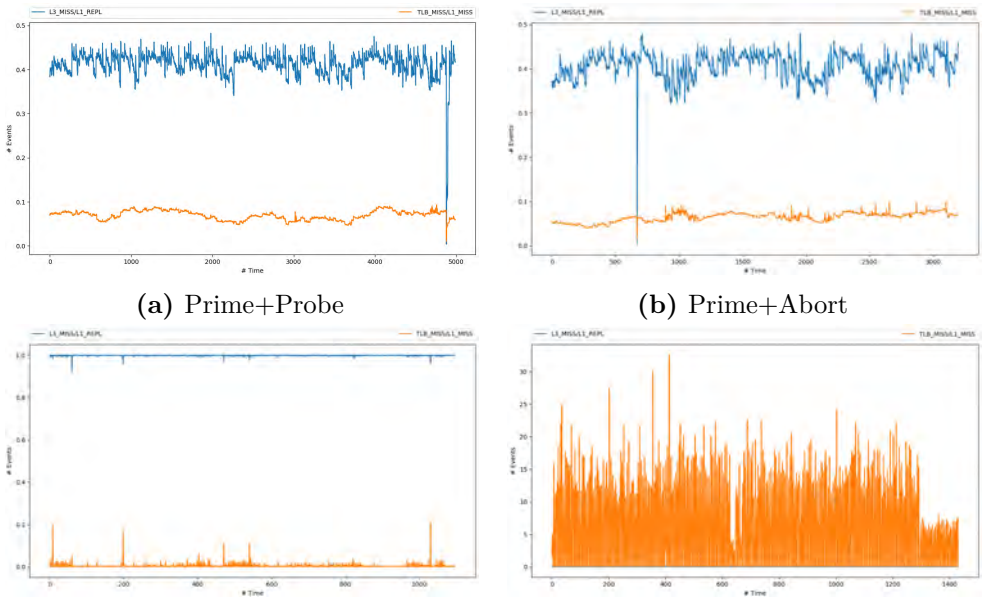
(c) Non-Malicious Application: Correctly Classified (Canneal)

Figure 4.15. Second Metric Results:
■ MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT
■ DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS

As we can see from the examples shown in Figure 4.15, by introducing the second metric it is possible to classify (Figure 4.15c) as non-malicious application which was erroneously classified as malicious by using only the first metric (a false positive).

4.4.3 Experimental Classification Results

Several tests were run on malicious and non-malicious applications to assess the accuracy and precision of the established metrics. The graphs from these experiments are shown in Figures 4.16 and 4.17.



(a) Prime+Probe **(b)** Prime+Abort

(c) Flush+Reload **(d)** Flush+Flush

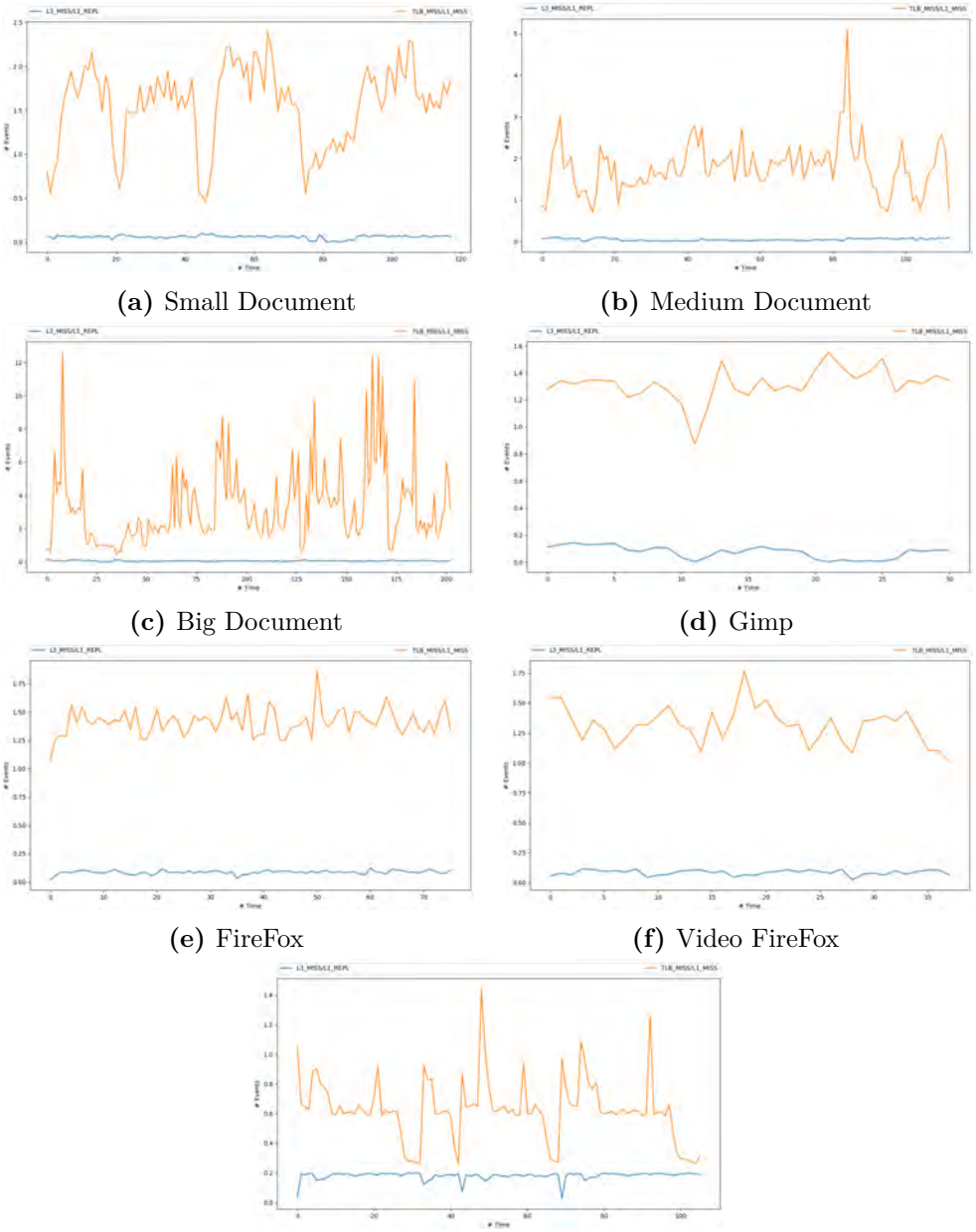
Figure 4.16. Malicious Applications Metrics Results:
MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT
DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS

By the results in Figure 4.16, we can see that in most cases malicious applications have higher values for the first metric (the blue line) than the values for the second metric (the orange line). This suggests that the memory area used during the execution of the program is very limited, because the value of the second metric is very close to 0, but that the application is forced to frequently access main memory. This is reflected int the fact that replacements in the first level of cache correspond to L3 misses, which is indicated by the fact that the value of the first metric is very high (in some cases it approaches to 1).

The consideration just given cannot be applied to the case of *Flush+Flush*, which generated results completely different from the others. This is because the discussed metrics are based on the idea that this family of attacks and attack techniques repeatedly access the cache, provoking many cache misses, in order to obtain the secret information. Rather, *Flush+Flush* using *cflush* as the main operation reduces significantly cache misses and cache accesses.

In the case of non-malicious applications (shown in Figure 4.17) the results of the metrics are perfectly specular.

We can deduce, by observing that the second metric has a value considerably greater than 1, that the number of TLB misses is greater than the number of



(a) Small Document **(b)** Medium Document

(c) Big Document **(d)** Gimp

(e) FireFox **(f)** Video FireFox

(g) VLC

Figure 4.17. Non-Malicious Applications Metrics Results:
■ MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT
■ DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS

L1 misses. Thus, they succeed of exploiting the principle of locality despite the estimated number of virtual to physical translations loaded in TLB is considerable. Furthermore, the value acquired by the first metric deviates slightly from 0 in all the tests that have been conducted. This implies that the number of L3 misses is lower than that of L1D replacements. Consequently, applications are not forced to access the main memory frequently, and the number of lines in the first cache level

is consistent, which suggests that the cache is not invalidated.

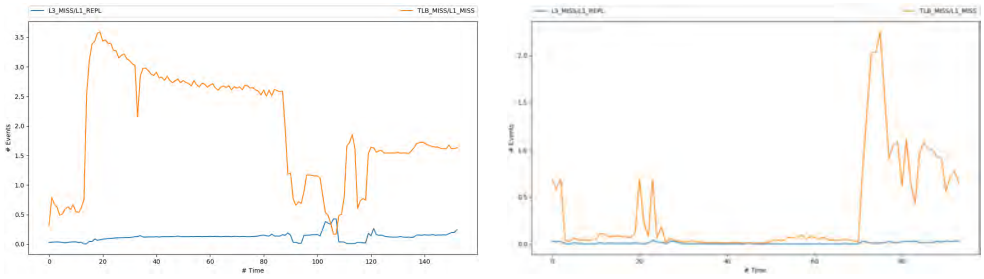
The tests conducted so far are on commonly used programs that have mixed behaviors. We decided to conduct tests that focus on specific behaviors, which can be adopted by various applications, so that these behaviors cannot be influenced and mitigated by one another. For the class of CPU-bound application, we relied on the Princeton Application Repository for Shared-Memory Computers (PARSEC), a benchmark suite composed of multithreaded programs. The PARSEC distribution consists of PARSEC packages and frameworks. The packages correspond to reference programs, libraries and other essential components. Each package can be compiled in different ways, as determined by a build configuration. The build configurations contain information such as the functionality of the package to be enabled, the compilers to be used and the way the package is to be optimized. PARSEC is supplied with predefined inputs that can be used to execute benchmarks. The inputs for each program have different characteristics such as the execution time and the working set size [2]. The benchmarks from the PARSEC Benchmarks Suite are:

- *Canneal*: implements a Simulated Annealing (SA) algorithm using to simulate some problems in chip design. SA belongs to the class of the local searches algorithm which aim to find a local optimum over a big search space. This application uses sophisticated lock free synchronization techniques and enforces its execution via a cache aware design.

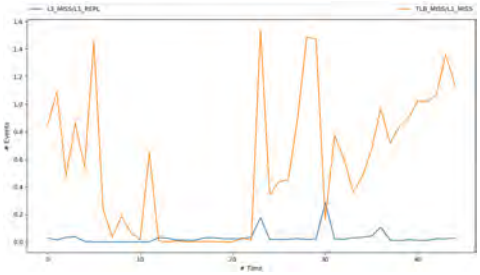
- *Freqmine*: uses an array-based version of the FP (Frequent Pattern-growth) growth method for Frequent Item set Mining (FIMI). It is an Intel RMS benchmark originally developed by Concordia University. *freqmine* has been included in the PARSEC benchmark suite due to the increasing use of data mining techniques.

- *Splash-2x*: is a benchmark that includes applications and kernels mainly in the area of high performance computing (HPC). It has been widely used to evaluate multiprocessors and their projects. The new version of *Splash-2* is called *Splash-2x* because it also has different input data sets on different scales.

As can be seen from the results shown in Figures 4.18 and 4.19, the combination of the two metrics classifies the monitored applications as non-malicious. These applications are intended to isolate specific behaviors and, in the particular case of CPU-bound ones, to resemble as much as possible the attacks we are trying to detect, which make intensive use of the cache. The similarity can be noted by the values assumed by the second metric, which suggests that the number of TLB misses is less than the number of L1 misses, which could represent a suspicious behavior. However, when analyzing the second metric we can observe that the value is constantly around zero. This result indicates that the applications are able to exploit the locality principle, since the number of L3 misses is much lower than the number of L1D replacements. In conclusion, the metrics chosen are valid even when the system is subjected to a high workload.

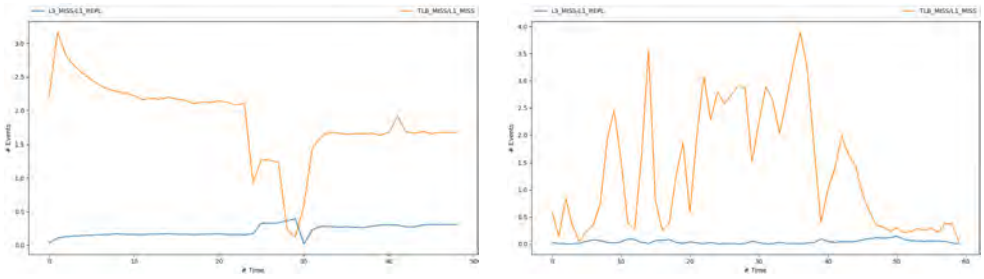


(a) Canneal **(b)** Freqmine

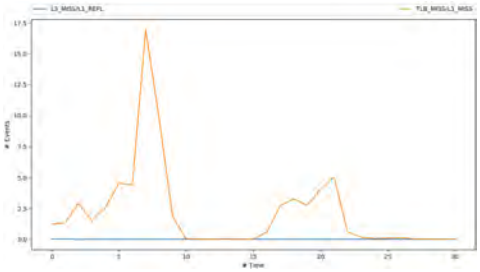


(c) Splash-2x

Figure 4.18. Results using *simsmall* as input;
■ MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT
■ DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS



(a) Canneal **(b)** Freqmine



(c) Splash-2x

Figure 4.19. Results using *simlarge* as input;
■ MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT
■ DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS

Finally, to make the tests performed on non-malicious applications more realistic, a preamble and a conclusion were added to executables who implemented the

attack techniques. The addition of these two fragments is intended to mitigate the oscillations caused by the attack techniques. This artifice could create problems if aggregate metrics or an extremely basic sampling frequency were used, which are on an informative level like aggregate metrics, as shown in the example shown in Figure 4.20.

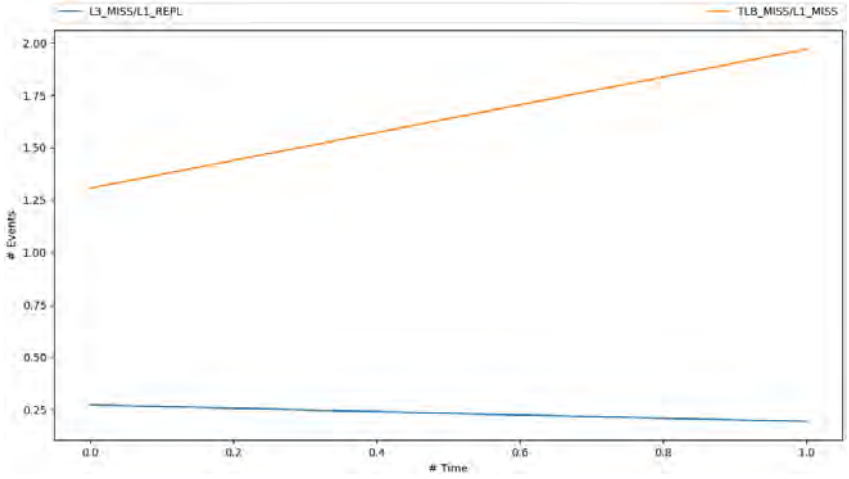
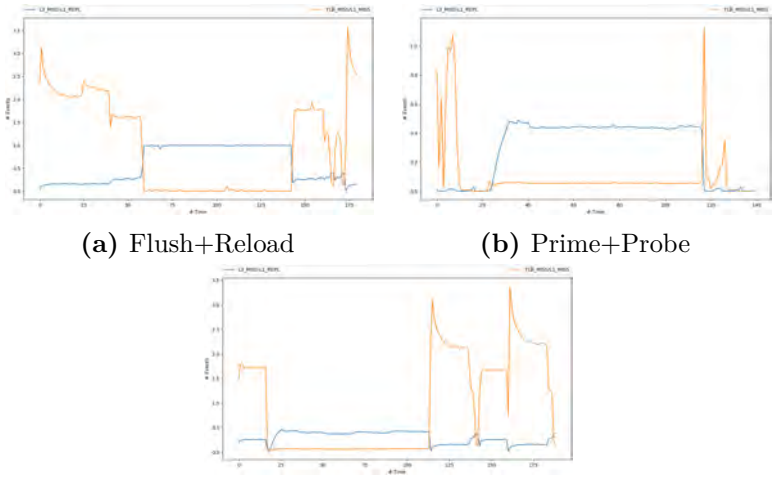


Figure 4.20. Example of malicious application with preamble and conclusion monitored at low frequency

Because the values obtained from the metrics, are mitigated by parts of non-malicious code, they may not undergo such a high variation so as to allow the distinction between a malicious and a non-malicious application.

Instead, with the metrics and the chosen frequency we obtain the results shown in Figure 4.21.



(a) Flush+Reload **(b)** Prime+Probe

(c) Prime+Abort

Figure 4.21. Results malicious application with preamble and conclusion;
■ MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT
■ DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS

In the plots it is easy to distinguish non-malicious code, the preamble and the conclusion, from the attack techniques. In conclusion we have no false negatives due to mitigation. The tests were conducted on the three attack techniques that we are able to monitor even without preamble and conclusion. *Flush+Flush* was not included in the study because it has a different behavior and cannot be identified with the chosen metrics.

4.4.4 Automatic Classification

Once we understand how distinguish malicious applications from non-malicious ones, we found a method to automatically classify them. Doing so, we can avoid to analyze each graph, but the number of false positives and the number of false negatives could increase. Since the difference is given by the position of the two plotted metrics, we compute the value of:

$$\frac{\texttt{MEM\_LOAD\_RETIRED\_L3\_MISS/L1D\_REPLACEMENT}}{\texttt{DTLB\_LOAD\_MISSES\_STLB\_HITS/MEM\_LOAD\_RETIRED\_L1\_MISS}}$$

This value is computed for each sample, and if this value is below a certain threshold then it can indicate suspicious behavior.

To determine this threshold, two results were evaluated, one to represent malicious applications and one for non-malicious applications, which differed most from the results obtained by the respective classes. So the two graphs considered are that of *Prime+Probe*, in Figure 4.22, and that of *Canneal* using *simsmall* as input, in Figure 4.23.

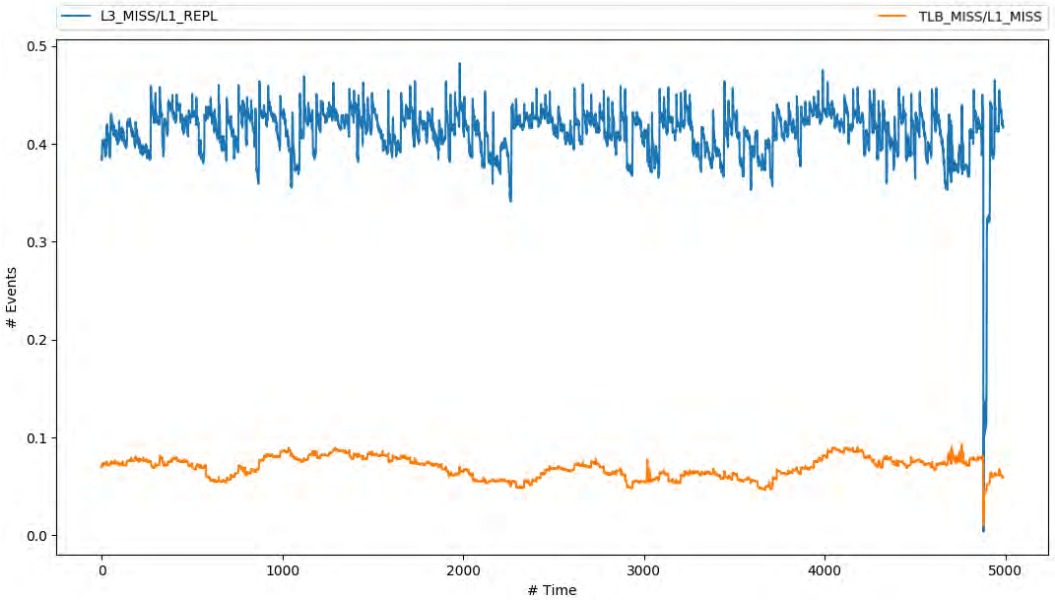


Figure 4.22. Prime+Probe

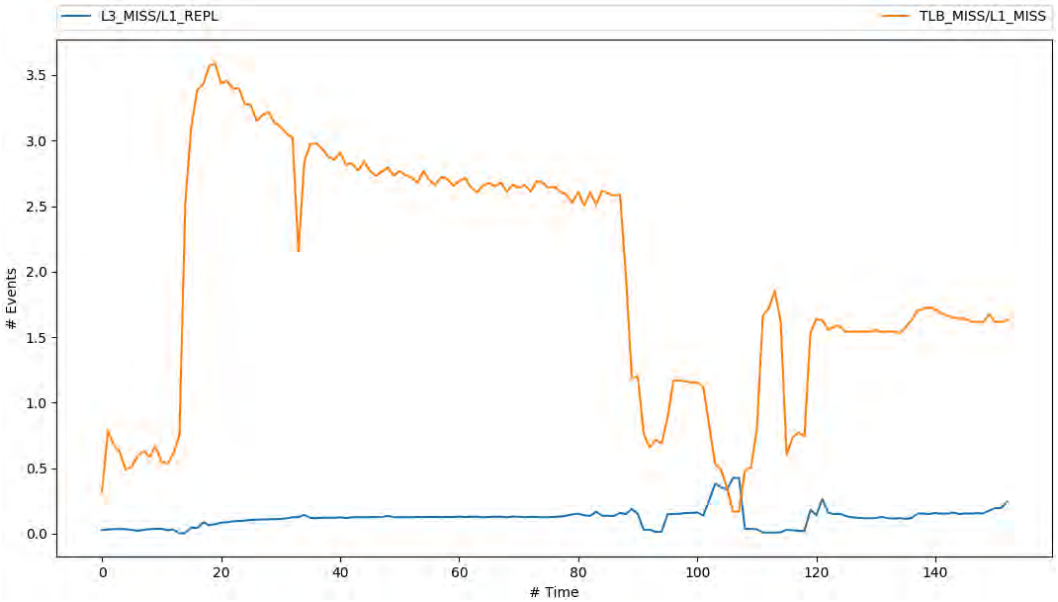


Figure 4.23. Canneal

In particular, for *Prime+Probe* the value was taken from the point where the two graphs are closest, without considering the samples saved at the beginning and at the end of the monitoring because those values could be due to a preamble and a conclusion. The value was taken at about time 1000 and the value of `MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT` is 0.356, while that of `DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS` is 0.097, so the result of the fraction is 0.272. While as far as *Canneal* is concerned, the value was calculated at the point where the graph most closely resembles those of malicious applications, i.e. where the two curves intersect and change the relative position, thus around the time 110. The value of `MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT` is 0.489 , while that of `DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS` is 0.252, so the result of the fraction is 0.515.

These results correspond to the most extreme results that can be obtained by analyzing the tests performed, in fact all malicious applications have a value less than or equal to that obtained by *Prime+Probe* and non-malicious applications have a value greater than or equal to that obtained by *Canneal*.

Following this, we chose a threshold such that if the value obtained from the fraction of the two metrics is less than this threshold then the behavior of the application is considered as suspect, otherwise if the value is above the threshold it is considered a signal of normal behavior.

However, we cannot classify as malicious any application that gets a value below the threshold because the number of false positives could become too high. To overcome this problem, we have devised a heuristic procedure which consists in assigning a score to the behavior of the application. This score will vary during execution as follows:

- increases the score by α in case the value obtained from the fraction of the

two metrics is lower than the threshold

- decrement of φ (where φ could also be equal to α) the score if the value obtained from the fraction of the two metrics is higher than the threshold

The score can take a value between 0 and *maximum value*. If the score reaches *maximum value* then the application is classified as malicious, otherwise it is classified as non-malicious. In this way the attacks are identified after a limited number of samples and it is not necessary to examine the whole execution, while the non-malicious applications should not be wrongly classified because the points in which the application has a suspicious behavior, in most cases they are sporadic and distant from each other so the score is diminished thanks to normal behavior before it reaches the *maximum value*.

4.5 Related Work

There are other proposal in the literature that use PMCs to detect attacks, some of which are discussed in this section. Unlike the study proposed in this thesis, the other studies have more information available to understand if an attack is underway or not. Some of them focus on a single attack technique, doing so there is the possibility of being able to study the technique in more detail, monitoring specific events of the technical one. This was not possible in the proposed methodology because it was designed to detect the family of side-channel attacks and therefore the events had to be as general as possible to understand most of the attacks while limiting false positives at the same time. Another very important aspect is that most proposal in the literature monitor both the execution of the attacker and that of the victim. By doing this, it is possible to obtain more data to be analyzed because the code executed by the victim, influenced by the execution of the attack, tends to accentuate the frequency variations of the monitored events that are detected when an application is under attack. On the other hand, this approach severely limits the applicability of these results in the real environment.

The morks most similar to the methodology presented here are those reported in the papers: *Flush+Flush: A Fast and Stealthy Cache Attack* [7] and *NIGHTs-WATCH: A Cache-Based Side-Channel Intrusion Detector using Hardware Performance Counters* [23], which we discuss in the following.

The idea presented in the paper *Flush+Flush: A Fast and Stealthy Cache Attack* [7] is based on the usage of the *perf* Linux profiler. The tool is accessible from command line and provides the possibility to use a number of parameters in order to customize the profiling that the tool will have to perform. perf is powerful: it can instrument CPU performance counters and can rely on other techniques that allow you to perform a dynamic break of a kernel function so as to collect debug information and performance data. Using this tool, it is possible to carry out lightweight profiling. In [7], the authors profile both the attacker and the victim to increase the accuracy of their results, since even from the execution of the victim relevant information for the detection can be collected. The events monitored, listed in Figure 4.24, range over all aspects of a program so as to have a better overview of its behavior.

Name	Description
BPU_RA/_RM	Branch prediction unit read accesses/misses
BRANCH_INSTRUCTIONS/_MISSES	Retired branch instructions/mispredictions
BUS_CYCLES	Bus cycles
CACHE_MISSES/_REFERENCES	Last-level cache misses/references
UNC_CBO_CACHE_LOOKUP	C-Box events incl. `clflush` (all slices)
CPU_CYCLES/REF_CPU_CYCLES	CPU cycles with/without scaling
DTLB_RA/_RM/_WA/_WM	Data TLB read/write accesses/misses
INSTRUCTIONS	Retired instructions
ITLB_RA/_RM	Instruction TLB read/write accesses
L1D_RA/_RM/_WA/_WM	L1 data cache read/write accesses/misses
L1I_RM	L1 instruction cache read misses
LL_RA/_WA	Last-level cache read/write accesses

Figure 4.24. List of hardware performance events monitored

After selecting the events, they monitored the system under different conditions: under normal conditions, under stress (for example loop reading and writing in dynamically allocated 256MB arrays), running common applications, such as Twitter or FireFox, and under attack. After which they compared the results obtained from the different executions trying to understand which were the differences between a system under attack and a normal execution. Finally they defined the metric shown in Figure 4.25.

$$\frac{C_{\texttt{CACHE\_MISSES}}}{C_{\texttt{ITLB\_RA}} + C_{\texttt{ITLB\_WA}}} \geq k_m, \quad \text{or} \quad \frac{C_{\texttt{CACHE\_REFERENCES}}}{C_{\texttt{ITLB\_RA}} + C_{\texttt{ITLB\_WA}}} \geq k_r$$

Figure 4.25. Detection Metrics

The parameters used correspond to:

- *CACHE_MISSES*: occur when a data is accessed after it has been flushed from the Last Level Cache.
- *CACHE_REFERENCES*: occur when the cache is accessed regardless of whether there is a hit or a miss.
- *(ITLB_RA and ITLB_WA)*: the two counters are normalized using the number of accesses to the TLB obtained from the sum of accesses in read mode and in write mode.
- k: following the experiments the two thresholds, k_m and k_r, are defined (one for each metric) and if the value obtained exceeds the threshold then with a certain degree of confidence it can be declared that the system is under attack.

Using these metrics it is possible to detect some of the side-channel attacks, but not all of them. In fact, it is not possible to detect *flush+flush* due to the absence of memory accesses from this attack. The same tried to use the event *UNC_CBO_-CACHE_LOOKUP*, which counts the number of references to the cache, but the quantity of false positives was not negligible. At the same time, we cover the same malicious applications, although by using different (more general) metrics.

The proposal presented in the paper *NIGHTs-WATCH: A Cache-Based Side-Channel Intrusion Detector using Hardware Performance Counters* [23] tries to combine the use of Machine Learning with Hardware Performance Counters in order to detect Side-Channel attacks. To perform the profiling, the authors used a tool called *Performance API (PAPI)*. The PAPI project specifies a standard API (Application Programming Interface) to access the hardware performance counters available on most modern microprocessors. PAPI offers two interfaces to exploit the underlying hardware counters; a simple and high level interface for the acquisition of simple measurements and a completely programmable low level interface for users with more sophisticated needs. PAPI can be divided into two levels of software. The upper level consists of the API support functions and machine independent. The lower level defines and exports a machine-independent interface for machine-dependent functions and data structures. In this way, it provides portability across different platforms [28].

Before performing the profiling, the authors had to decide which events to profile and what to profile. Most of the selected events are related to the cache, in particular the number of L1, L2 and L3 hits and misses and in addition there are also the number of core cycles and the number of retired branch instructions that were mispredicted by the processor. Unlike the methodology proposed in this thesis, also in this paper, both the attacker and the victim are profiled, thus obtaining more information from the results of the counters. The methodology used by them can be divided into three parts: *training*, *execution* and *detection*.

In the *training* phase the Machine Learning Models conceived are trained using a large number of already labeled samples. A sample is the set of results obtained by monitoring through PAPI the events listed in the introduction of a system under attack or not. The labels used for the classification are only two: *Attack* and *No Attack*.

In the *execution* phase, the system is monitored, which may be under attack or not, and new samples are run at runtime.

A the end, in the *detection* phase, the samples collected in the previous phase are analyzed by the trained Machine Learning Models. Each model classifies the sample into on of the two categories (*Attack* and *No Attack*) and through these results it is determined whether the system is under attack or not.

The results reported were obtained under different load conditions of the system and using different models. The accuracy of the classification depends on how well the models used have been trained, while the overhead depends on the loading conditions and the frequency with which the samples are collected. Furthermore, for every possible attack scenario a new model must be devised, which must be trained with appropriate samples, making the entire detection process quite expensive. Moreover, the need for a training phase makes this proposal less suitable for possibly new attacks and general applicability, which is one of the explicit goal of this thesis.

Chapter 5

Reference Implementation

As a reference implementation, we have developed a Linux Kernel Module for Linux x86-64, which exploits the Program Monitoring Counters (PMCs) technology by Intel. This module implements the methodology which we have discussed in the previous Chapter.

Our implementation is compatible with the 6th, 7th and 8th generation of Intel core processors, based respectively on the Skylake, Kaby Lake and Coffee Lake microarchitectures. This is due to the fact that the module directly works on model-specific registers (MSRs) which most of the time are specific for each processor model and each processor has its own list of supported architectural performance monitoring events.

The module has been designed to monitor the system in order to collect information about different aspects of the behavior of a thread or a process. It is possible to decide which events must be taken into consideration to generate statistics, which are then used to decide whether some process should be considered as malicious or not.

The Intel support used to acquire the information needed is implemented at the firmware level. For this reason, the overhead that is added to program execution is low.

We have carried out stress tests to study the capability of our reference implementation to detect hardware attacks, such as Meltdown, Specter and Foreshadow. However, the use of the tool is not limited only to this purpose, in fact there are a large number of events that can be monitored and a large number of options that can be exploited. For example, the tool could be used to detect other types of attacks, once appropriate strategy and the corresponding metrics to detect it are identified.

5.1 Module Organization

When the module is loaded, it performs the following operations:

- First of all, since MSRs are specific for each processor model, it checks whether the current operating system is running on an Intel Fam and ensures that the processor generation is compatible with the implementation. Once ensured that the underlying system is supported, the module verifies that all the necessary

supports (PMC and PEBS) are available, and that the format of PEBS records is consistent with the implementation.

- After disabling all the PMC (both fixed and programmable) that may have remained set by a previous execution, the module setups and registers a new Non-Maskable Interrupt[7] (NMI) handler. The handler is employed to manage any interrupt generated by the aforementioned support. Inside the interrupt handler, controls are used to distinguish the possible sources.

- A new char device is created and registered in order to handle the interaction between the kernel space and the user space through the ioctl commands. To invoke ioctl commands of a device, the user-space program would open the registered char device first, then send the appropriate ioctl() and any necessary arguments.

- At the end, the module allocates the needed data structures for PEBS, such as the debug store area and the pools of buffers to collect the PEBS samples and the tasklet information. The structures will then be used to efficiently save the samples, which otherwise would be lost or overwritten, useful for future analysis or to generate statistics.

5.2 IOCTL commands

ioctl (input/output control) is a system call for device specific input/output control operations that cannot be expressed through normal system calls [20]. The kernel is designed to be extensible, and accepts extra modules called a *device drivers*. Device drivers runs directly in kernel space and exposes through the *ioctl* interface a system call that allows user space to communicate with it. Through this system call, the device can offer an arbitrary number of functions (each with its unique identification number), allowing the extension to be programmed without adding system calls to the operating system. To call one of these functions, the user will use the system call of the device and pass it 3 arguments: the file descriptor of the opened device pseudofile, the identification number of the function and an argument, which is of type unsigned long. In this way it is also possible to pass a pointer to any type of data structure.

The *ioctl* operations, which we have implemented, can be divided into three groups: configuration and control of the PMCs, management of the processes to be profiled and presentation of the results obtained. To obtain the arguments of the individual functions, a number of parameters have been defined, that are passed to a command line utility which we have implemented as part of this work. This utility directly interacts with the Kernel Module, to carry out the monitoring activities which we have previously discussed.

[7] A Non-Maskable interrupt is a hardware interrupt that cannot be ignored by standard masking techniques.

5.2.1 PMCs Configuration

The first operation to active our system is to configure the PMCs. Based on the monitoring activities that must be performed, you can choose the configuration that suits you best. The choices that can be made in order to have this configuration are the following: the number of PMCs active on the single CPU and which with PEBS, the mode (user and/or kernel) in which profiling must be active, the list of events that must be monitored by the PMCs and, finally, the value from which they must start to count the PMCs and the one to which they must be reset in case of overflow.

To indicate the number of active PMCs on each CPU, a hexadecimal value is entered for each available CPU. This value is converted so as to have a sequence of four bytes (one for PMC). If the x-th bit is 1, then the PMCx on that CPU must be activated, otherwise if it is 0, then it will remain off. The number of PMCs using PEBS and their monitoring mode are managed in a similar way. If a PMC of a specific CPU is turned off, the values concerning PEBS and the monitoring mode are ignored. Each PMC must be configured to monitor a particular event. So, for each PMC, a hexadecimal value is passed, which will then be written to the register *IA32_PERFEVTSELx MSR* (shown in Figure 3.1). The value is composed of the *Event Select* and the *Unit Mask*, which are reported in *Intel 64 and IA-32 Architectures Software Developer's Manual* [9]. Finally, it is possible to decide the value from which the PMCs will start to count, and also the one from which they would start counting again if they reach their maximum value. The default value from which they start counting and with which they are reset is 0 and if not specified otherwise it will be used. The start value is written to the register *IA32_PMCx* (described in the Section 3.1.1) before the PMCs are turned on. Instead, the writing of the reset value is managed inside an *interrupt handler*[8] which is activated following an interrupt caused by the achievement of the maximum value of a PMC. The information about the configuration of the PMCs (as used by the Kernel Module) is saved inside a variable size array of metadata, defined as the *configuration_t* type. The size is variable because there is a structure for each available CPU present on the machine on which the code is running. The *configuration_t* is organized as in Listing 5.1:

```
1 typedef struct{
2     int valid;
3     pmc_conf_t pmcs[MAX_PMC];
4 }configuration_t;
```

Listing 5.1. `configuration_t` structure

The first field, *valid*, is use for optimization reasons. If none of the PMCs on this CPU must be activated, *valid* is equal to 0 and there is no need to analyze the second field, *pmcs*. The *pmcs* fields is a fixed size array (the size is saved into the *MAX_PMC* macro and is equal to the maximum number of PMCs available for a single CPU), of *pmc_conf_t* structures, organized as in Listing 5.2.

[8] An interrupt handler, also known as an interrupt service routine or ISR, is a special block of code associated with a specific interrupt condition.

```
1    typedef struct{
2        int valid;
3        uint64_t event;
4        uint64_t start_value;
5        uint64_t reset_value;
6        int user;
7        int kernel;
8        int enable_PEBS;
9    }pmc_conf_t;
```

Listing 5.2. `pmc_conf_t` structure

The *valid* filed has the same purpose as the one present in the *configuration_t* structure, in such a way there is no need to analyze the further fields. The remaining fields contain the basic information to configure the PMCs obtained as described at the begin of this section.

Once the configuration has been established, it must be passed as parameter to the Kernel Module. The Kernel Module and the command line utility share the same header file and therefore the same *configuration_t* structure definition, so the configuration can be passed without further changes or conversions. The configuration can be used both to setup and to reset the selected PMCs. In order to do this, there are two *ioctl* commands: *SETUP_PMC* and *RESET_PMC*. At the end of the execution of one of these commands, the PMCs are not activated or deactivated. In this way it is possible to change the configuration of a single PMC without having to rewrite the entire configuration, and it is also possible to setup and/or reset a PMC during the monitoring phase without having to turn off, and then turn back on, the support and without altering the collected data.

5.2.2 Processes Management

According to the proposed methodology all the processes running in the system are monitored simultaneously and for each of these processes the corresponding data structures to store the collected samples will be allocated and will be managed on context switch. However, for the sole purpose of being able to run the tests necessary to choose the metrics and verify their goodness, the possibility of choosing a single process to monitor was added. The registration of the process, that must be monitored, can be while the application is running or it can be finalized before the process starts to run. In the first case, an external application, like *htop*, is used to retrieve the PID of the running process and it is passed to the main function through the command line. Otherwise, it is possible to pass to the main function the name of the application or the *relative path*[9] of the executable and, if necessary, the parameters that would be passed when the application or executable starts.

```
1    int pid = fork();
2    if (pid == 0) {
3        ioctl(fd, ADD_TID, getpid())
```

[9] A relative path is a way to specify the location of a directory or a file relative to the present working directory (pwd) [3].

```
4           ioctl(fd, PID_PROFILER_ON)
5           execvp(name,args);
6           return 0;
7       }
8       else {
9           waitpid(pid, NULL, 0);
10      }
```

Listing 5.3. Add an application to profile

As shown in Listing 5.3, once you pass this information to the command line utility, the main function executes *fork()*. *fork()* creates a new process by duplicating the calling process.The new process is called the *child* process. The calling process is called the *parent* process [14]. The two processes are distinguished by the value assumed by the variable pid. The child process adds the PID just followed in order to be profiled and then activates the profiling. Finally it executes an *exec()*. The *exec()* family of functions replaces the current process image with a new process image [13]. In this way the child process will execute the application that was passed from the command line. The parent process waits for the child to finish, otherwise we would not know when it stop running and then when to retrieve the monitoring results.

Both for the direct addition of the PID and the one made using the *fork-exec* technique just described, the PID of the process to be monitored must be passed from user space as parameter to an *ioctl* function, name *ADD_PID*. Once the PID is register, it is possible to turn on and turn off the profiler using respectively the *PID_PROFILER_ON* and *PID_PROFILER_OFF* functions.

5.2.3 Data Retrieved and Post Processing

Once the application is terminated and the monitoring operation is stopped, it is possible to extract the obtained results.

If PEBS is active, the *ioctl* command used to retrieve the results is *READ_BUFFER*, that returns a buffer of variable size. The size is obtained before calling the *READ_BUFFER* function, using the additional function *SIZE_BUFFER*. The function returns an unsigned long containing the number of samples stored by PEBS during the monitoring. The basic element of this buffer is the *pebs_struct* structure. The structure is composed of the same fields as the record saved by the *PEBS assist*, reported in Figure 3.8. The obtained buffer can be printed on screen or can be saved on a *.dat* files for further analysis.

In Figure 5.1, an example of the usage of PEBS and its results are shown. In this case, PEBS is used to collect information about the memory load operations performed by a specific application. The monitoring is performed with different values of *start value* and *reset value*. The results are the memory load operations sampled by PEBS. The graph shown the frequency (*start value* and *reset value*) on the abscissa and the ratio between the number of load operations sampled and the number of actual load operations on the ordinate. The ratio represents the quality of the support at different frequencies.

Instead, if PMCs are used as standard counters, the results can be retrieved

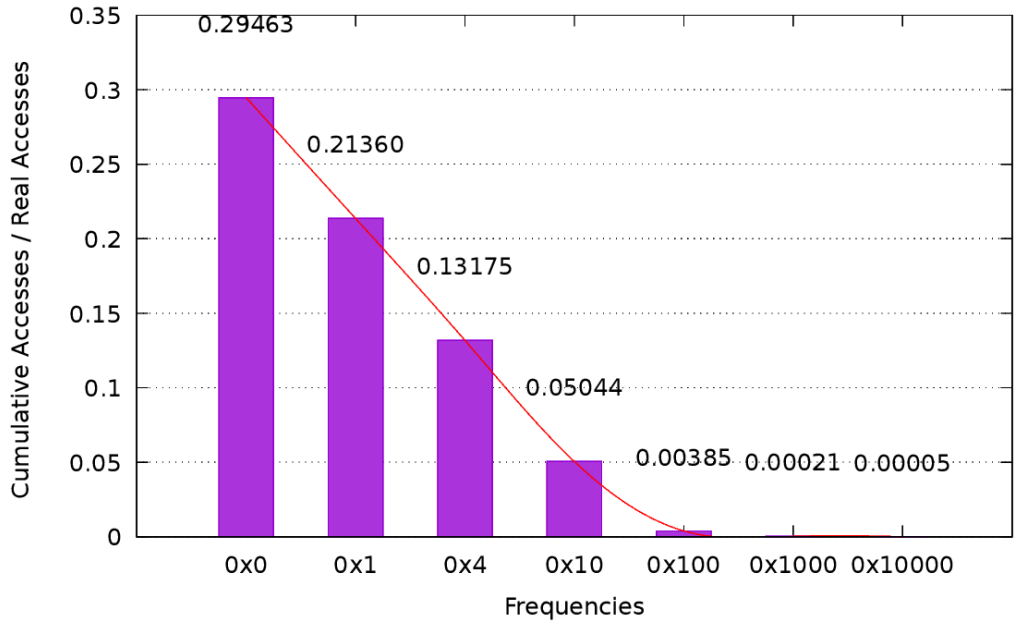


Figure 5.1. Graphic example obtained with PEBS

using the *ioctl* command named *EVT_STATS*. The function returns an array of *statistics* structure (shown in Listing 5.4) of variable size.

```
1    struct statistics{
2        uint64_t fixed0;
3        uint64_t fixed1;
4        struct event_stat events[MAX_ID_PMC];
5    };
```

Listing 5.4. `statistics` structure

The first two field of the structure contain the number of events registered by the two fixed PMCs (*Fixed PMC0* and *Fixed PMC1*). To store the result of the other PMCs (*Programmable PMCs* and *Offcore PMCs*), we define a new structure, named *event_stat*, shown in Listing 5.5.

```
1    struct event_stat{
2        uint64_t event;
3        uint64_t stat;
4    };
```

Listing 5.5. `event_stat` structure

The structure reports the number of events counted by the PMC and which event was monitored by it. This last field is not necessary for the *Fixed PMCs* because they can monitor only one event.

In order to allocate an array of the correct size to contain the results, the main function calls *SIZE_STAT* command before *READ_BUFFER*, that returns an unsigned long containing the number of samples present in the buffer.

We have used *matplotlib.pyplot*[10], to plot the retrieved data. An example is shown in Figure 5.2.

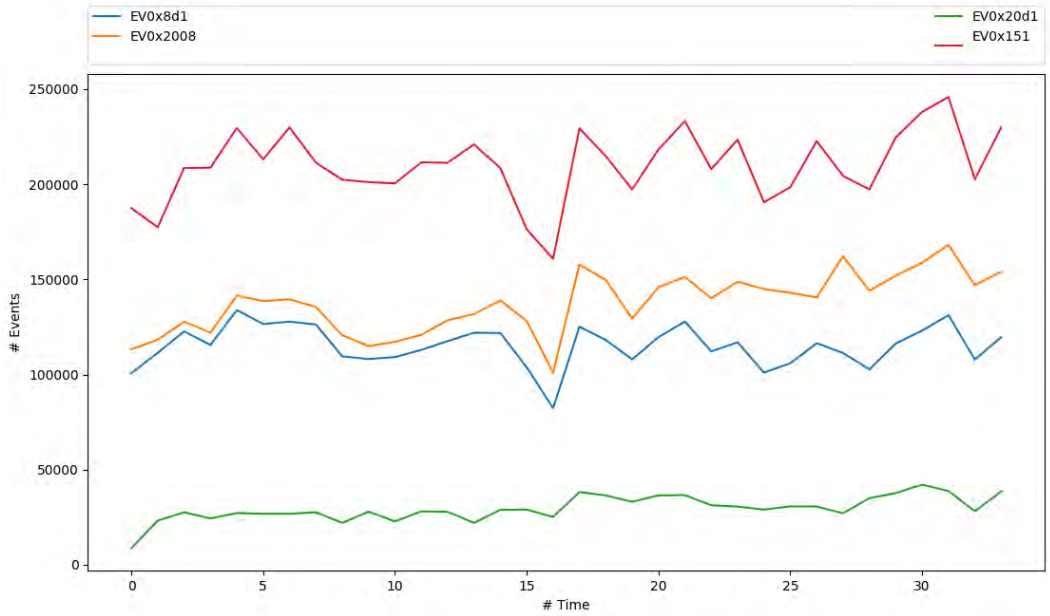


Figure 5.2. Graphic example obtained with PMCs

The plot shows the number of core cycles (*Fixed PMC1*) on the x axis and the number of events counted by the single PMCs on the y axis.

5.3 Hooking into the scheduler

To manage the monitoring of applications at runtime, our Kernel Module must be informed of what is the currently running application on a given CPU core. To this end, we install a *kretprobe post-handler* (shown in Listing 5.6) hooked to the function responsible of the *context switch*.

```
1     if (!profiled_on)
2         goto off;
3     start_monitoring(current->pid)
4         enablePMCS();
5         goto end;
6 off:
7     disablePMCS();
8 end:
9     return 0;
```

Listing 5.6. Context Switch Post-Handler Kretprobe

The post-handler function, as the name suggests, is executed after the execution of the context switch. It checks whether the profiler is turned on, picks process to

[10] *matplotlib.pyplot* is a state-based interface to *matplotlib*. It provides a lot of functionalities to plot graphs and diagrams [12].

be monitored. If the profiler is not turned on all the PMCs (that could have been previously activated) are disabled. Otherwise, we activate the PMCs.

5.4 Handling PMC Overflow

As mentioned, when a PMC reaches its maximum value, it overflows. If the *APIC interrupt enable* bit (presented in Figure 3.1) is enabled, an interrupt is fired. Two possibilities for handling the interrupt have been examined. The first possibility is to modify the native linux Non-Maskable Interrupt (NMI) handler, since the interrupt generated by the PMC is an NMI. The modified handler should check upon any fired NMI whether it was generated by the PMCs or not. The second possibility is to add a new entry to the Interrupt Descriptor Table (IDT), and associate it with a custom handler triggered by the PMCs. We adopted this second solution, as it is much cleaner. The interrupt descriptor table (IDT) links each interrupt with a descriptor, from which the address of the routine used to service the request is extracted by the firmware, as seen in the Figure 5.3.

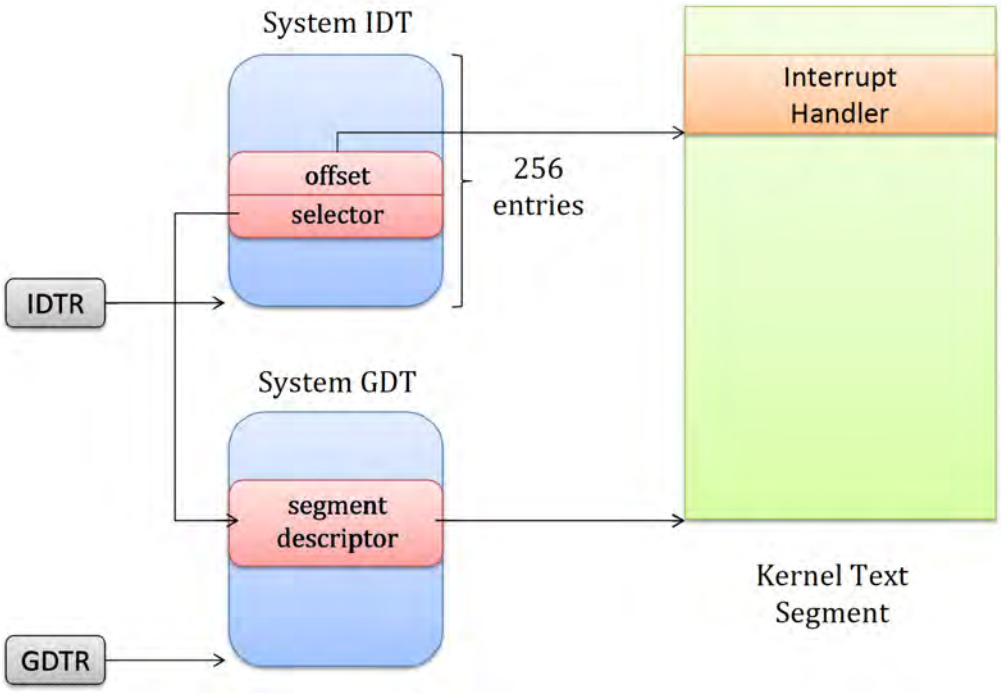


Figure 5.3. Interrupt Descriptor Table

Each entry is referred as an *interrupt vector*. The *interrupt vector* is composed by an *offset* and a *selector*. The *selector* is used to identify a *segment descriptor*, which contains a pointer to the memory segment that holds the interrupt handler to be executed. To determine the final starting address of the handler, the firmware applies the *offset* contained in the *interrupt vector* to the base address stored in the *segment descriptor*.

The handler installed by our module is used to collect the samples obtained

during monitoring phase which will then have to be analyzed to understand the behavior of the program. To set the sampling frequency, the chosen reset value is negated and is written into the *Fixed PMC1* registry, that counts the number of core cycles (in this way, after a number of events equal to the reset value, it will reach the maximum value). When the PMC1 Fixed reaches the maximum value, the values of the active PMCs are saved in the data structure shown in Listing 5.4. This structure is added to the *Linked List* which is returned to the user at the end of the monitoring. At the end of the interrupt handler all the PMCs are reset to zero to start a new sampling, while the Fixed PMC1 is set to reach its maximum value after a number of events equal to the reset value.

5.5 Overhead

We calculated the overhead that is added to the application during monitoring. This result is very important because if the overhead is too high, even if the methodology is valid because it produces useful information, it could render the monitored application unusable due to the delay that would be introduced.

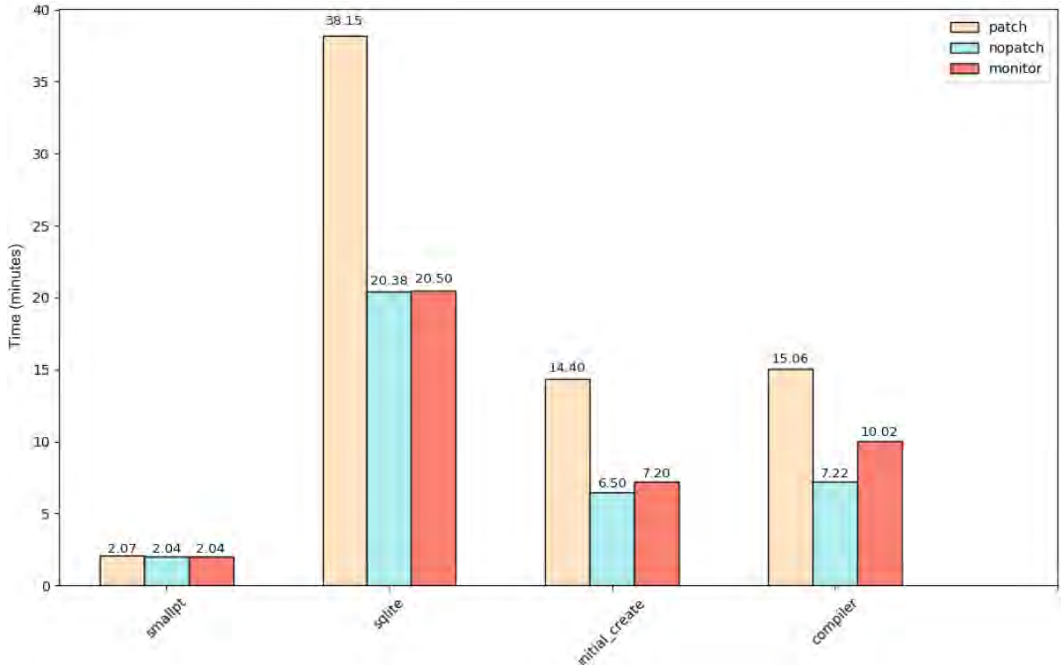


Figure 5.4. Overhead Report

The data, shown in Figure 5.4, were collected by using the *Phoronix Test Suite*. The *Phoronix Test Suite* is a complete test and benchmarking platform available that provides an extensible framework so you can easily add new tests. The software is designed to effectively perform qualitative and quantitative benchmarks in a clean, reproducible and easy to use way. The Phoronix Test Suite can be used to simply compare computer performance, which is the functionality for which we are using it, or can be used within the organization for internal quality assurance, hardware and software validation and continuous integration/performance management [1]. The

benchmarks from the Phoronix Test Suite, which we have used for our overhead assessment, are:

- *smallpt*. The peculiarities of the SmallPT benchmark are its different workload and the fact that it is written in less than 100 lines of C++ code. This benchmark returns an image using a Monte Carlo algorithm and presents the execution time (in seconds) as a response variable.
- *sqlite*. This is a simple benchmark of SQLite. This test measures the time to perform a pre-defined number of insertions on an indexed database. The database is implemented with a single database file. Access to the database is done with file locking.
- *compilebench*. Compilebench tries to extend a filesystem by simulating some of the disk I/O common in creating, compiling, stating, patching and reading kernel trees. It indirectly measures the file systems performance regarding the maintenance of directory locality as the disk fills up and directories age. The test is configured to simulate the creation of 10 initial directories using the make -j command, this is called *makej mode*. The I/O considered in the tests are: *initial create* and *compiler*.

We have chosen these benchmarks to conduct tests that focus on various behaviors that a malicious application or not can adopt. In particular we selected *smallpt* because it is a CPU-Bound benchmark and therefore, given the nature of the analyzed attacks, their behavior was simulated. While the other three benchmarks use different syscall[11] profiles, forcing the application to make a substantial number of context switches. Given the implementation of software patches, we should have an influence on the performance of the considered applications.

These benchmarks were run in three different scenarios to evaluate the performance of the module described with the software patches adopted by current systems to prevent side channel attacks. The scenarios are as follows:

- *Kernel 4.18*, which has activated software patches to mitigate the attacks discussed in this thesis, plotted with the label *patch*.
- *Kernel 4.9*, which does not provide for any countermeasure for side channel attacks, plotted with the label *nopatch*.
- *Kernel 4.9* with the addition of the kernel module just described, plotted with the label *monitor*.

The results met our expectations and clearly show that the added overhead of monitoring is minimal with respect to the scenario that does not involve any countermeasure, and, above all, with respect to the added overhead of using patched software.

[11] A *syscall* is the mechanism, used by a user-level or application-level process, to request a kernel-level service from the operating system

Benchmark	Overhead Patch	Overhead Monitor
smallpt	1.015x	1x
sqlite	1.872x	1.005x
initial create	2.215x	1.107x
compiler	2.086x	1.387x

Table 5.1. Overhead Results

Using the results shown in Table 5.1, we calculated the average overhead added by both the software patches and the kernel module. There is an average increase of 1.796x in the application execution time when the pacthes software is active against the 1,124x obtained from the monitoring carried out by the module.

In conclusion, it must be considered that once it has been established that an application is not malicious, the monitoring for that particular application can be switched off via the module, thus reducing the overall overhead. In contrast, software patches are constantly active for all applications that are currently running in the system.

Chapter 6

Conclusions and Future Work

In this thesis we presented a new methodology to detect hardware-based side channel attack. The main idea of this proposal is to use hardware support provided by Intel processors (Performance Monitoring Counters) to obtain information about applications execution. The support is not used according to its original purpose, which is to monitor the system to gather information on the resources usage, but it is used to detect cache-based side channel attacks. The presented methodology uses this support to get an overview of the behavior of monitored applications, in particular focusing on how applications use the cache. We selected a group of events to be monitored suitable for the intended purpose. However, these events cannot be directly compared because they could represent different behaviors of the monitored applications. For example, a high number of misses at the first level of the cache can mean both that an application is trying to perform a side channel attack and that the data processed by the non-malicious application does not allow the exploitation of the locality principle because they occupy a large area of memory. For this reason, the events were grouped together to obtain appropriate metrics, so that it was possible to compare the results deriving from the monitoring of different malicious and non-malicious applications. The metrics achieved by the theoretical and empirical studies are `MEM_LOAD_RETIRED_L3_MISS/L1D_REPLACEMENT` and `DTLB_LOAD_MISSES_STLB_HITS/MEM_LOAD_RETIRED_L1_MISS` and they have led to very promising results considering also that unlike other results in the literature, it is not necessary to monitor the victims of the attacks. In this way, we are required to collect less information, but in doing so we have a more realistic analysis given that during a real attack we are not given to know who the victim is. To make the detection of the attacks automatic and less dispensary, a procedure was designed that examines the relationship of the two metrics. If the value is lower than an automatically computed score, then a suspicious behavior is detected, otherwise the application is considered non-malicious. To reduce the number of false positives, the score is tuned at runtime accounting for suspicious execution patterns. If the score exceeds a predetermined maximum value, then the application is classified as malicious. The overhead introduced by this monitoring system is low, although it depends on the frequency with which a new sample, containing the value of the monitored metrics, is generated. To obtain a correct result, it is necessary to work at a moderate frequency otherwise the samples will contain meaningless information. In

case the of high frequency, a sample collects too few information and it is non possible to distinguish a malicious application from a non-malicious one. Instead, in case of low frequency, if the attack is mitigated through a preamble and/or a conclusion, the number of false negative grows too much. The low overhead is certainly the biggest difference with the software patches presented in the introduction of this thesis and which are considered the de facto solution to side channel attacks. In fact, software patches added such a high overhead to application execution also in cases in which no attack at all is carried.

We have already planned several improvements for the future version of this methodology:

- *Processor Event Based Sampling*: events were monitored using standard sampling offered by PMCs, but PEBS could also be used to detect cache-based side-channel attacks. As already mentioned in Section 3.7, PEBS also provides additional information that could be very useful for analyzing the behavior of the monitored application. In particular, given that the methodology aims to study the use of the cache by monitored applications, the PEBS record fields can be exploited to obtain more precise information regarding load and store operations, such as the linear address of the source of the load, or linear address of the destination of the store. Furthermore, through the PEBS, *Data Source/-Store Status* record field, it is possible to know if a given load operation caused a TLB miss or not. Therefore there is no need to infer information through the number of events counted by *DTLB_LOAD_MISSES_STLB_HIT*, but it is possible to obtain more precise information.

- *more precise time window*: the time window in the exposed methodology is determined using the *Fixed PMC1*. This PMC monitors the event CPU_-CLK_UNHALTED_THREAD_ANY, that counts the number of core cycles while the core is not in a halt state. We plan to introduce a time window based on the real execution time of the monitored application and an automatic management of it. The execution time would constitute a more precise sampling unit, less subject to errors. A time window that is managed automatically may be preferable to intensify the observation during a specific execution period and keep it as low as possible in other situations so as to further decrease the overhead added while monitoring. Two limit values should be determined to manage the time window, so that the monitoring frequency is never too hich or too low.

- *Compatibility*: it would be interesting to explore other microarchitectures, different from the one studied in this thesis. Each microarchitecture has its own set of events that can be monitored, so the obtainable metrics could be different from those identified and analyzed in Section 4.25. It would be possible to determine if there are metrics, which were not feasible on the considered microarchitecture, more suitable for the purpose given and then assess whether it is possible to further increase the accuracy.

- *Dynamic monitoring*: currently it is possible to monitor up to four events, excluding the Fixed PMCs. We plan to design a mechanism that, based on

the execution of the application, determines at runtime which are the most appropriate events to monitor in order to understand if the application is malicious or not. This mechanism would further improve the accuracy, because we could identify specific metrics (or even dynamic ones) for a single type of attack and we will no longer have to look for the common factor among all the attacks. Therefore, thanks to this optimization, it could be useful to study Flush + Flush and find metrics that adapt to its peculiarities that distinguish it from other applications.

List of Figures

Listings

Bibliography

[1] Openbenchmarking. Accessed: 2019-07-09. Available from: https://openbenchmarking.org/.

[2] The parsec benchmark suite. Accessed: 2019-06-28. Available from: https://parsec.cs.princeton.edu/documentation.htm.

[3] Linux.com | the source for linux information, absolute path vs relative path in linux/unix (July 21, 2017). Accessed: 2019-06-05. Available from: https://www.linux.com/blog/absolute-path-vs-relative-path-linuxunix.

[4] Admin. L1d replacement percentage. Accessed: 2019-06-27. Available from: https://software.intel.com/en-us/vtune-amplifier-help-l1d-replacement-percentage.

[5] Dinis, N. Cache why level it. In *Proceedings of the 3rd Internal Conference on Computer Architecture, Universidade do Minho* (2002).

[6] Disselkoen, C., Kohlbrenner, D., Porter, L., and Tullsen, D. Prime+ abort: A timer-free high-precision l3 cache attack using intel {TSX}. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 51–67 (2017).

[7] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+ flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 279–299. Springer (2016).

[8] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 897–912 (2015).

[9] Guide, P. Intel® 64 and ia-32 architectures software developerâĂŹs manual. *Volume 3B: System programming Guide, Part*, **2** (2011).

[10] IBM. Transactional memory. Accessed: 2019-05-31. Available from: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm.

[11] Intel, R. Intel® 64 and ia-32 architectures optimization reference manual. *Intel Corporation, Sept*, (2014).

[12] John Hunter, D. D. and Eric Firing, M. D. matplotlib (2012). Accessed: 2019-06-04. Available from: `https://matplotlib.org/api/pyplot_api.html`.

[13] Kerrisk, M. Exec (2017). Accessed: 2019-06-05. Available from: `http://man7.org/linux/man-pages/man3/exec.3.htm`.

[14] Kerrisk, M. Fork (2017). Accessed: 2019-06-05. Available from: `http://man7.org/linux/man-pages/man2/fork.2.html`.

[15] Kocher, P., et al. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, (2018).

[16] Koruyeh, E. M., Khasawneh, K. N., Song, C., and Abu-Ghazaleh, N. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)* (2018).

[17] Lipp, M., et al. Meltdown. *arXiv preprint arXiv:1801.01207*, (2018).

[18] Lipp, M., et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 973–990 (2018).

[19] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pp. 605–622. IEEE (2015).

[20] Madieu, J. *Linux Device Drivers Development: Develop customized drivers for embedded Linux* (October 20, 2017).

[21] Maisuradze, G. and Rossow, C. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2109–2122. ACM (2018).

[22] Mittal, S. A survey of techniques for architecting tlbs. *Concurrency and Computation: Practice and Experience*, **29** (2017), e4061.

[23] Mushtaq, M., Akram, A., Bhatti, M. K., Chaudhry, M., Lapotre, V., and Gogniat, G. Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, p. 1. ACM (2018).

[24] Mushtaq, M., Akram, A., Bhatti, M. K., Rais, R. N. B., Lapotre, V., and Gogniat, G. Run-time detection of prime + probe side-channel attack on aes encryption algorithm. In *2018 Global Information Infrastructure and Networking Symposium (GIIS)*, pp. 1–5. IEEE (2018).

[25] Sarda, P., Motwani, G., and Patil, D. Evaluation of tlb prefetching techniques. *Downloaded on Mar*, **20** (2009), 1.

[26] Software, I. Understanding the instruction pipeline. Accessed: 2019-05-31. Available from: `https://techdecoded.intel.io/resources/understanding-the-instruction-pipeline/#gs.fwcnbj`.

[27] Software, I. Measuring instruction latency and throughput (2008). Accessed: 2019-05-31. Available from: https://software.intel.com/en-us/articles/measuring-instruction-latency-and-throughput.

[28] Terpstra, D., Jagode, H., You, H., and Dongarra, J. Performance application programming interface (2010). Accessed: 2019-06-06. Available from: https://icl.utk.edu/papi/overview/index.html.

[29] Van Bulck, J., et al. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 991–1008 (2018).

[30] VUSec. xlate. https://github.com/vusec/xlate (Aug 17, 2018).

[31] Weisse, O., et al. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. Tech. rep., Technical report (2018).

[32] Yarom, Y. and Falkner, K. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 719–732 (2014).

[33] Yoo, R. M., Hughes, C. J., Lai, K., and Rajwar, R. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 19. ACM (2013).

[34] Zheng, Y., Davis, B. T., and Jordan, M. Performance evaluation of exclusive cache hierarchies. In *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004.*, pp. 89–96. IEEE (2004).